

The logo features the text "SocketTools 10" in a bold, sans-serif font. "SocketTools" is in blue, and "10" is in orange. Below it, "Library Edition" is written in a smaller, blue, sans-serif font. The text is centered over a faint, light gray background graphic of a globe with latitude and longitude lines.

SocketTools 10

Library Edition

Developer's Guide and Technical Reference
Version 10.0.1468.2518

Introduction

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

The SocketTools Library Edition includes support for the industry standard Transport Security Layer (TLS) and Secure Shell (SSH) protocols which are used to ensure that data exchanged between the local system and a server is secure and encrypted. The Library Edition implements the major secure protocols such as HTTPS, FTPS, SFTP, SMTPS, POP3S, IMAPS and more. Your data is protected with TLS 1.2 using 256-bit encryption and full support for client certificates. SocketTools also includes an FTP and HTTP server API, as well as a general purpose TCP server API that can be used to create custom server applications. There's no need for you to understand the details of certificate management, data encryption or how the security protocols work. All it takes is a few lines of code to enable the security features, and SocketTools handles the rest.

The following are just some of the features in the SocketTools 10 Library Edition:

- Support for Windows 10, Windows Server 2019 and Visual Studio 2019
- Standard Windows dynamic link libraries (DLLs) with no external dependencies
- A comprehensive API with more than 20 libraries and over 950 functions
- Support for both synchronous and asynchronous network connections
- Includes libraries that can be used to create custom client and server applications
- Provides cloud-based application storage and geographical IP location services
- Support for the standard TLS 1.2 protocol with 256-bit AES encryption
- Support for both implicit and explicit TLS connections
- Support for the SSH protocol and integrated support for SFTP as part of the FTP API
- Support for standard and secure proxy servers using FTP and HTTP
- Support for using client and server certificates in PKCS12 format
- Thread-safe implementation with full support for multithreaded applications
- An extensive Developer's Guide and online Technical Reference
- Easy redistribution for any number of applications and end users

Developer's Guide

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

Technical Reference

The Technical Reference provides extensive documentation on all of the functions in each of the SocketTools libraries. It's here that you'll find information on how a function should be called, what the

arguments are and what options are available. If it is your first time using a particular library, we recommend that you first read the overview of that library in the Developer's Guide.

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

SocketTools Licensing Information

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

Evaluation Licenses

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

Runtime Licensing

When you install SocketTools with a serial number, a runtime license key will be automatically generated for you and stored in a file named `csrtkey10.h` in the Include folder where you've installed the product. There are similarly named files for other languages, such as `csrtkey10.bas` for Visual Basic and `csrtkey10.pas` for Delphi. These files define the SocketTools runtime licensing key which must be passed to the Initialize function in the library that you are using. If you are using a language that does not have a license key already defined for it, you can create a text file that contains the license key using the License Manager utility. More information about that utility is provided below.

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

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

License Manager

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

SocketTools 10 Upgrade Information

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

Supported Platforms

SocketTools 10 is supported on Windows 7, Windows Server 2008 R2 and later versions. Earlier versions of the operating system, including Windows 2000, Windows XP and Windows Vista are no longer supported by Microsoft and are not recommended for use with SocketTools.

Developers who are redistributing applications which target Windows 10 or Windows Server 2019 should upgrade to ensure compatibility with the platform and current development tools. Secure connections on Windows XP and Windows Vista has been deprecated because those platforms do not support TLS 1.2 and most services will no longer accept connections from a client using SSL 3.0 or TLS 1.0.

Development Tools

As standard Windows dynamic link libraries, SocketTools 10 may be used with virtually any programming language which can call exported functions in a DLL, either by name or by ordinal. Import libraries are provided for Visual C++ in both x86 and x64 COFF format, and for Borland's C++ compilers in OMF32 and ELF64 format. Other languages should use the conventions appropriate for calling an exported function, such as the Declare statement in Visual Basic. Although the libraries may be used with .NET languages, it is recommended you use the SocketTools .NET Edition if you are creating applications for the .NET Framework.

SocketTools Header Files

In SocketTools 10 all of the library constants, functions and C++ classes are in a single header file named **cstools10.h**. This header file also includes the error codes that are defined in a separate header file named **cserror10.h**. There are three special macros that can be defined to control how applications are built using the SocketTools libraries:

CSTOOLS_NO_LIBRARIES

Visual C++ and Borland C++ supports a pragma which can be used to automatically specify the names of libraries that the compiler should attempt to link with in order to resolve function calls. By default, it is not necessary to explicitly specify the names of the SocketTools import libraries to link to. However, if you wish to explicitly link to specific import libraries, then define this macro prior to including the cstools10.h header files.

CSTOOLS_NO_NAMESPACE

When compiling a C++ application, the SocketTools functions are defined in a namespace called SocketTools. This prevents the possibility of conflicts with functions of the same name that may be used in other libraries. If you prefer the functions to be defined in the global namespace instead, then define this macro prior to including the cstools10.h header file.

CSTOOLS_NO_CLASSES

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

SocketTools C++ Class Wrappers

The C++ class wrappers for the SocketTools libraries have been moved from a separate file into the cstools10.h header file, and are now automatically included whenever a C++ program is compiled. The

classes have been designed for compatibility with a variety of C++ compilers; however, there are certain features which are only available if the application is compiled using the Microsoft Foundation Classes (MFC) or Active Template Library (ATL). For example, if an application is built using MFC, each class is derived from CObject and there are additional overloaded methods implemented which support the use of objects like CString. If the application is built without using MFC, or a different C++ compiler is used, those methods will not be available.

Upgrading From SocketTools 9.5

If you are upgrading from version 9.5, applications will be source code compatible with the SocketTools 10 libraries. In most cases, all you will need to do is install the current version, update the header file and import library references, and then recompile your application. Note that the library file names have changed, as have the export function ordinals. If your build projects use the **SocketTools9** environment variable to specify the location of the header files and import libraries, you need to change it to use **SocketTools10**. This environment variable is automatically defined during the installation process.

The SocketTools 10 libraries are not binary compatible with the SocketTools 9.5 libraries and cannot be used as drop-in replacements. If you have declared functions by ordinal in your application (something not typically done), those values have changed and must be updated. If you have declared the exported functions by name, you will be able to reference the new libraries using the same names.

Your runtime license key has changed for SocketTools 10, which will require you to define the new key in your application when calling the initialization functions for each API. As with previous versions of SocketTools, you can use the License Manager utility to generate a file which contains the runtime key you should use. The SocketTools 9.5 and earlier runtime license keys are not valid for the version 10 libraries and an error will be returned if an invalid runtime key is specified.

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

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

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

- The name of the primary header file is **cstools10.h** and the **cserror10.h** header file defines the error code values. The **csrtkey10.h** header file defines the runtime license key for this version of SocketTools. The runtime license key is normally generated automatically when SocketTools is installed with a valid serial number. It can also be generated using the License Manager utility installed with SocketTools.
- When using C or C++, the **cstools10.h** header file should be included before **Windows.h** to ensure that the compiler does not try to include the deprecated Windows Sockets 1.1 header file. Failure to do so can result in compiler warnings about duplicated macros and function prototypes.
- If you are using the Embarcadero C++ compiler, additional checks are imposed to ensure the SocketTools header files are included before **WinInet.h** and **Vcl.h** files. This prevents conflicts between our APIs and function prototypes defined in those header files.
- The name for each of the DLLs and import libraries has been changed with the new version. Applications written using previous versions of SocketTools must be updated to use the new file

names.

- Some constant values have changed and added. This will not impact applications that used the defined macros, but may impact any application which uses hard-coded numeric values, rather than constants or macros.

Applications which continue to use SocketTools 9 and earlier libraries can be installed side-by-side with the version 10 libraries. It is recommended that you redistribute the libraries in the same folder as your application executable, rather than a shared system folder such as C:\Windows\SysWOW64 or C:\Windows\System32. It is not recommended that you attempt to use different versions of the libraries within the same application.

Upgrading From SocketTools 8.0

If you are upgrading from version 8.0 or an earlier version, most applications will be source-compatible with version 10 and will not require significant changes to existing code. The only requirement is to include the new header files and link to the new version of the import libraries. Note that the library file names have changed, as have the export function ordinals. If your build projects use the **SocketTools8** environment variable to specify the location of the header files and import libraries, you need to change it to use **SocketTools10**. This environment variable is automatically defined during the installation process.

The SocketTools 10 libraries are not binary compatible with the version 8 libraries and cannot be used as drop-in replacements. If you have declared functions by ordinal in your application (something that is not commonly done), those values have changed and must be updated. If you have declared the exported functions by name, you will be able to reference the new libraries using the same values.

The runtime license key has changed for SocketTools 10, which will require you to define the new key in your application when calling the library's initialization function. As with previous versions of SocketTools, you can use the License Manager utility to generate a new header file which contains the runtime key you should use. The version 8.0 runtime key is not valid for the current libraries and an error will be returned if an invalid runtime key is specified.

Applications which continue to use SocketTools 8.0 and earlier libraries can be installed side-by-side with the version 10 libraries. It is recommended that you redistribute the libraries in the same folder as your application executable, rather than a shared system folder such as C:\Windows\SysWOW64 or C:\Windows\System32. It is not recommended that you attempt to use different versions of the libraries within the same application.

SocketTools 8.0 and earlier versions established secure connections using versions of TLS which are now considered deprecated. Most services today will reject connection attempts from clients who request TLS 1.0 or SSL 3.0. SocketTools 10 uses TLS 1.2 by default for all secure connections, however some older servers which have not been updated may reject the connection attempt. It is possible to force SocketTools 10 to use an earlier version of TLS for backwards compatibility by explicitly setting the *dwProtocol* member of the SECURITYCREDENTIALS structure, specifying the protocol version required.

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

Most of the networking APIs have a new option to force the library to establish an IPv6 network connection. By default, the libraries will still give preference to using IPv4 by default, since it is still the dominant version of the protocol in use today. It is important to note, however, that if you force your application to use only an IPv6 connection, your program may not work on many end-user systems.

Library File Names

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

File Name	Import Library	Description
csdnsv10.dll	csdnsv10.lib	Domain Name Service Library
csftpv10.dll	csftpv10.lib	File Transfer Protocol Library
csftsv10.dll	csftsv10.lib	File Transfer Server Library
cshtpv10.dll	cshtpv10.lib	Hypertext Transfer Protocol Library
cshtsv10.dll	cshtsv10.lib	Hypertext Transfer Server Library
csicmv10.dll	csicmv10.lib	Internet Control Message Protocol Library
csmapv10.dll	csmapv10.lib	Internet Message Access Protocol Library
csmsgv10.dll	csmsgv10.lib	Mail Message Library
csmtpv10.dll	csmtpv10.lib	Simple Mail Transfer Protocol Library
csncdv10.dll	csncdv10.lib	File Encoding Library
csnvtv10.dll	csnvtv10.lib	Terminal Emulation Library
csnwsv10.dll	csnwsv10.lib	Network News Transfer Protocol Library
cspopv10.dll	cspopv10.lib	Post Office Protocol Library
csrshv10.dll	csrshv10.lib	Remote Command Protocol Library
csrsv10.dll	csrsv10.lib	Syndicated News Feed Library
cstimv10.dll	cstimv10.lib	Time Protocol Library
cstntv10.dll	cstntv10.lib	Telnet Protocol Library
cstshv10.dll	cstshv10.lib	Secure Shell Protocol Library
cstxtv10.dll	cstxtv10.lib	Text Messaging Library
cswebv10.dll	cswebv10.lib	Web Services Library
cswhov10.dll	cswhov10.lib	Whois Protocol Library
cswskv10.dll	cswskv10.lib	Windows Sockets (SocketWrench) Library

SocketTools Evaluation License

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

Redistribution Restrictions

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

Runtime Licensing

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

For more information, refer to the [Licensing](#) and [Library Initialization](#) sections.

SocketTools Library Redistribution

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

System Requirements

SocketTools is supported on Windows 32-bit and 64-bit desktop and server platforms. The minimum required desktop platform is Windows 7 with Service Pack 1 (SP1) installed. The minimum required server platform is Windows Server 2008 R2 with Service Pack 1 (SP1) installed. It is recommended that the current service pack be installed for the operating system, along with the latest Windows updates available from Microsoft. Some features may require Windows 8.1 or later versions of the platform. When this is the case, it will be noted in the documentation.

Windows 2000, Windows XP and Windows Vista are no longer supported by Microsoft. SocketTools is designed for Windows 7 as the minimum operating system version and may not work correctly on earlier versions of Windows.

Library Redistribution

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

If SocketTools is installed on a 64-bit version of Windows, the 32-bit DLLs are installed in the C:\Windows\Syswow64 folder and the 64-bit DLLs are installed in the C:\Windows\System32 folder. When redistributing one or more of the libraries, it is important to make sure that you are selecting the correct version, which is determined by the development tool used and the target platform. For example, if you are using Visual Studio 2010 and target the Win32 platform, then you should only redistribute the 32-bit DLL, regardless if the target system is the 32-bit or 64-bit version of Windows. This is because 32-bit programs can only reference 32-bit libraries. When the application is installed on 64-bit Windows, it will be executed by the WoW64 subsystem which provides a 32-bit environment for the application.

Version Information

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

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

Installation Directory

It is recommended that you install the SocketTools libraries in the same folder with the application that uses them. It is not recommended that you install the libraries under the Windows system folder on an end-user system. If you choose to install the libraries in the Windows system folder, you must ensure that the installer makes the appropriate registry entries to indicate that they are shared files. Failure to do so can result in

the libraries being removed if the user uninstalls your application, which may cause other applications to fail.

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

Windows Install Packages

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

Package Name	Description
cstools10_library_x86.msi	SocketTools 10 redistributable libraries for 32-bit applications. This installer is what developers should use if they are targeting the x86 platform and want their software to run on both 32-bit and 64-bit versions of Windows.
cstools10_library_x64.msi	SocketTools 10 redistributable libraries for 64-bit applications. This installer should only be used if 64-bit development tools were used to build the application, and can only be installed on 64-bit versions of Windows.

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

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

```
msiexec /qn /I cstools10_library_x86.msi
```

For the complete list of command line options for **msiexec**, refer to the [Windows Dev Center](#) documentation.

Technical Support

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

Standard Support

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

[Frequently Asked Questions](#)

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

[Knowledge Base](#)

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

[Online Documentation](#)

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

[Release Notes](#)

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

Priority Support

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

Premium Support

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

License Agreement

This License Agreement is a legal agreement between you, either as an individual or a single entity ("Developer"), and Catalyst Development Corporation ("Catalyst") for the software product identified as "SocketTools Library Edition" ("Software" or "Software Product"). The Software Product includes executable programs, redistributable modules, controls, and dynamic link libraries ("Components" or "Software Components"), electronic documentation, and may include associated media and printed materials.

Installing this Software Product on to a hard disk or any other storage device of a computer, or loading any of the Components into the memory of any computer, constitutes use of the Software and shall acknowledge your acceptance of the terms and conditions of this License Agreement and your agreement to bound thereby.

1. GRANT OF LICENSE

Catalyst Development grants you as an individual, a personal, non-exclusive, non-transferable license to install the Software Product using an authorized serial number. If you are an entity, Catalyst grants you the right to appoint an individual within your organization to use and administer the Software Product subject to the same restrictions enforced on individual users. You may not network the Software or otherwise use it on more than one workstation or computer at the same time. Contact Catalyst for more information regarding multi-developer site licensing.

You may install the Software Product on one or more workstations or computers expressly for the purposes of evaluating the performance of the Software for a period of no more than thirty (30) days. If continued use of the Software is desired after the evaluation period has expired, then the Software Product must be purchased and/or registered with Catalyst Development for each computer or workstation. The Software Product must be removed from all unregistered workstations or computers after the evaluation period has expired.

2. COPYRIGHT

Except for the licenses granted by this agreement, all right, title, and interest in and to the Software Product (including, but not limited to, all copyrights in any executable programs, modules, controls, libraries, electronic documentation, text and example programs), any printed materials and copies of the Software Product are owned by Catalyst Development. The Software Product is protected by copyright laws and international treaty provisions. Therefore you must treat the Software Product like any other copyrighted material except that you may (i) make one copy of the Software solely for backup or archival purposes, or (ii) transfer the Software to a single hard disk, provided you keep the original solely for backup or archival purposes. You may not copy any printed materials that may accompany the Software Product. All rights not specifically granted in this Agreement, including Federal and International Copyrights, are reserved by Catalyst Development.

3. REDISTRIBUTION

(a) In addition to the rights granted in section 1, you are granted the right to use and modify those portions of the Software designated as "example code" for the sole purposes of designing, developing, and testing your software product, and to reproduce and distribute the example code, along with any modifications thereof, only in object code form, provided that you comply with section 3(c).

(b) In addition to the rights granted in section 1, you are granted a non-exclusive, royalty-free right to reproduce and distribute the object code version of any portion of the Software Product, along with any modifications thereof, in accordance with the above stated conditions.

(c) If you redistribute the sample code or redistributable components, you agree to: (i) distribute the redistributables in object code only, in conjunction with and as a part of a software application product developed by you which adds significant and primary functionality to the Software; (ii) not use Catalyst Development's name, logo, or trademarks to market your software application product; (iii) include a valid

copyright notice on your software product ; (iv) indemnify, hold harmless, and defend Catalyst Development from and against any claims or lawsuits, including attorney's fees, that arise or result from the use or distribution of your software application product; (v) not permit further distribution of the redistributables by your end user.

4. UPGRADES

If this copy of the Software is an upgrade from an earlier version of the Software, you must possess a valid full license to a copy of an earlier version of the Software to install and/or use this upgrade copy. You may continue to use each earlier version copy of the Software to which this upgrade copy relates on your computer after you receive this upgrade copy, provided that, (i) the upgrade copy and the earlier version copy are installed and/or used on the same computer only and the earlier version copy is not installed and/or used on any other computer; (ii) you comply with the terms and conditions of the earlier version's end user license agreement with respect to the installation and/or use of such earlier version copy; (iii) the earlier version copy or any copies thereof on any computer are not transferred to another computer unless all copies of this upgrade copy on such computer are also transferred to such other computer; and (iv) you acknowledge and agree that any obligation Catalyst may have to support and/or offer support for the earlier version of the Software may be ended upon availability of the upgrade.

5. LICENSE RESTRICTIONS

You may not rent, lease or transfer the Software. You may not reverse engineer, decompile or disassemble the Software, except to the extent applicable law expressly prohibits the foregoing restriction. You may not alter the contents of a hard drive or computer system to enable the use of the evaluation version of the Software for an aggregate period in excess of the evaluation period for one license. Without prejudice to any other rights, Catalyst Development may terminate this License Agreement if you fail to comply with the terms and conditions of the agreement. In such event, you must destroy all copies of the Software Product.

6. CONFIDENTIALITY

(a) The Software contains information or material which is proprietary to Catalyst Development ("Confidential Information"), which is not generally known other than by Catalyst, and which you may obtain knowledge of through, or as a result of the relationship established hereunder with Catalyst. Without limiting the generality of the foregoing, Confidential Information includes, but is not limited to, the following types of information, and other information of a similar nature (whether or not reduced to writing or still in development): designs, concepts, ideas, inventions, specifications, techniques, discoveries, models, data, object code, documentation, diagrams, flow charts, research, development, methodology, processes, procedures, know-how, new product or new technology information, strategies and development plans (including prospective trade names or trademarks).

(b) Such Confidential Information has been developed and obtained by Catalyst by the investment of significant time, effort and expense, and provides Catalyst with a significant competitive advantage in its business.

(c) You agree that you shall not make use of the Confidential Information for your own benefit or for the benefit of any person or entity other than Catalyst, except for the expressed purposes described in this section, in accordance with the provisions of this Agreement, and not for any other purpose.

(d) You agree to hold in confidence, and not to disclose or reveal to any person or entity, the Software, other related documentation, your product Serial Number or any other Confidential Information concerning the Software other than to such persons as Catalyst shall have specifically agreed in writing to utilize the Software for the furtherance of the expressed purposes described in this section, in accordance with the provisions of this Agreement, and not for any other purpose.

(e) You acknowledge the purpose of this section is to protect Catalyst Development's ability to limit the use of the data and the Software generally to licensees, and to prevent use of Confidential Information concerning the Software by other developers or vendors of software.

7. CONTINUATION OF SERVICE

Some features of the Software may require the use of remote servers under the control of Catalyst Development to provide specific services. Catalyst makes no warranty as to the availability of these services and reserves the right to discontinue these services at any time and without warning. These services may only be accessed using the Application Programming Interfaces (API) provided by the Software Product and access is limited to licensees and evaluation users of the Software.

We may suspend or terminate your access to these services without liability if (i) we reasonably believe that the services are being used (or have been or will be used) in violation of the Agreement, (ii) we reasonably believe that suspending or terminating your access is necessary to protect our network or our other customers, or (iii) the suspension or termination is required by law. We will give you reasonable advance notice of suspension or termination under this section and a chance to cure the grounds on which the suspension or termination is based, unless we determine, in our reasonable commercial judgment, that an immediate suspension or termination is necessary to protect Catalyst or its other customers from imminent and significant operational or security risk.

8. LIMITED WARRANTY

If within thirty days of your purchase of this software product, you become dissatisfied with the Software for any reason, you may return the software to Catalyst Development (or your dealer, if you did not purchase it directly from Catalyst) for a refund of your purchase price. To return the Software, you must contact Catalyst Development and obtain a Return Material Authorization (RMA) number. Catalyst will not accept returns of opened or installed software without an RMA number. Returns may be subject to the deduction from your purchase price of a restocking fee and all shipping costs.

CATALYST PROVIDES NO REMEDIES OR WARRANTIES, WHETHER EXPRESS OR IMPLIED, FOR ANY SAMPLE APPLICATION CODE, TRIAL VERSION AND THE NOT FOR RESALE VERSION OF THE SOFTWARE. ANY SAMPLE APPLICATION CODE, TRIAL VERSION AND THE NOT FOR RESALE VERSION OF THE SOFTWARE ARE PROVIDED "AS IS".

CATALYST DISCLAIMS ALL OTHER WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTY OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, WITH RESPECT TO THE SOFTWARE, THE ACCOMPANYING WRITTEN MATERIALS, AND ANY ACCOMPANYING HARDWARE.

9. LIMITATION OF LIABILITY

IN NO EVENT SHALL CATALYST OR ITS SUPPLIERS BE LIABLE FOR ANY DAMAGES WHATSOEVER (INCLUDING, WITH LIMITATION, INCIDENTAL, CONSEQUENTIAL, SPECIAL, OR EXEMPLARY DAMAGES OR LOST PROFITS, BUSINESS INTERRUPTION, OR OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OR INABILITY OF THIS CATALYST PRODUCT, EVEN IF CATALYST HAS BEEN ADVISED OF SUCH DAMAGES.

APART FROM THE FOREGOING LIMITED WARRANTY, THE SOFTWARE PROGRAMS ARE PROVIDED "AS-IS", WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED. THE ENTIRE RISK AS TO THE PERFORMANCE OF THE PROGRAMS IS WITH THE PURCHASER. CATALYST DOES NOT WARRANT THAT THE OPERATION OF THE PROGRAMS WILL BE UNINTERRUPTED OR ERROR-FREE. CATALYST ASSUMES NO RESPONSIBILITY OR LIABILITY OF ANY KIND FOR ERRORS IN THE PROGRAMS OR DOCUMENTATION, OF/FOR THE CONSEQUENCES OF ANY SUCH ERRORS. THE LAWS OF THE STATE OF CALIFORNIA GOVERN THIS AGREEMENT.

10. GOVERNMENT-RESTRICTED RIGHTS

United States Government Restricted Rights. The Software and related documentation are provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software - Restricted Rights at 48 CFR 52.227-19, as applicable. Manufacturer for such purposes is Catalyst Development Corporation,

11. EXPORT CONTROLS

You agree to comply with all relevant regulations, including but not limited to those, of the United States Department of Commerce and with the United States Export Administration Act to insure that the Software is not exported in violation of United States law. You acknowledge that the Software is subject to export regulations and agree that you will not export, re-export, import or transfer the software in violation of any United States or other applicable laws, whether directly or indirectly, and you will not assist or facilitate others in doing so. You acknowledge that you have the responsibility to obtain any export classifications and licenses as may be required to comply with such laws.

12. PROHIBITED DESTINATIONS

The exportation, re-exportation, sale or supply of Catalyst products, software components or documentation, directly or indirectly, from the United States or by a United States citizen wherever located, to Cuba, Iran, North Korea, Sudan, Syria, or any other country to which the United States has embargoed goods, is strictly prohibited without prior authorization by the United States Government. You represent and warrant that neither the United States Bureau of Export Administration nor any other federal agency has suspended, revoked or denied your export privileges. Catalyst products, software components or documentation may not be exported or re-exported to anyone on the United States Treasury Department's list of Specially Designated Nationals or the United States Department of Commerce Denied Person's List or Entity List.

13. GOVERNING LAW

This License is governed by the laws of the State of California, without reference to conflict of laws principles. Any controversy or claim arising out of or relating to this contract, or the breach thereof, shall be settled by arbitration administered by the American Arbitration Association ("AAA") under its Commercial Arbitration Rules, and judgment on the award rendered by the arbitrator(s) may be entered in any court having jurisdiction thereof. The arbitrator shall be a retired judge or attorney with at least 15 years commercial law experience and shall be selected either by mutual agreement of the parties or by AAA's selection process. The parties shall be entitled to take discovery in accordance with the provisions of the California Code of Civil Procedure, including but not limited to CCP §1283.05. The arbitration shall be held in San Bernardino, California and in rendering the award the arbitrator must apply the substantive law of the State of California.

14. GENERAL PROVISIONS

This License Agreement contains the complete agreement between the parties with respect to the subject matter hereof, and supersedes all prior or contemporaneous agreements or understandings, whether oral or written. You agree that any varying or additional terms contained in any purchase order or other written notification or document issued by you in relation to the Software licensed hereunder shall be of no effect. The failure or delay of Catalyst to exercise any of its rights under this Agreement or upon any breach of this Agreement shall not be deemed a waiver of those rights or of the breach.

If any provision of this agreement shall be held by a court of competent jurisdiction to be contrary to law, that provision will be enforced to the maximum extent permissible, and the remaining provisions of this agreement will remain in full force and effect.

SocketTools and other trademarks contained in the Software are trademarks or registered trademarks of Catalyst Development Corporation in the United States and/or other countries. Third party trademarks, trade names, product names and logos may be the trademarks or registered trademarks of their respective owners. You may not remove or alter any trademark, trade names, product names, logo, copyright or other proprietary notices, legends, symbols or labels in the Software.

SocketTools 10

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

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

Portions Copyright © 1993, 1994 The Regents of the University of California.

Portions Copyright © 1989 Massachusetts Institute of Technology.

Portions Copyright © 1995 Tatu Ylonen.

Portions Copyright © 1999, 2000 Neil Provos and Markus Friedl.

Portions Copyright © 1997, 2003 Simon Tatham.

Portions Copyright © 1995, 2005 Jean-loup Gailly and Mark Adler

Portions Copyright © 1991, 1992 RSA Data Security, Inc.

Information in this document is subject to change without notice. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Catalyst Development Corporation.

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

SocketTools 10 Library Edition Developers Guide

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

- Internet Control Message Protocol
- Internet Message Access Protocol
- Mail Message
- Network News Transfer Protocol
- News Feed
- Post Office Protocol
- Remote Command Protocol
- Simple Mail Transfer Protocol
- Telnet Protocol
- Terminal Emulation
- Text Messaging
- Time Protocol
- Whois Protocol
- Windows Sockets (SocketWrench)

SocketTools Features

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

Features of the SocketTools Library Edition include:

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

the server. The protocol negotiation, data encryption and decryption is handled transparently by the library. From the perspective of the application developer, it is just as if it were a standard connection to the server.

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

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

Getting Started

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

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

File Transfers

- [File Transfer Protocol](#)
- [Hypertext Transfer Protocol](#)

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

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

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

World Wide Web

- [Hypertext Transfer Protocol](#)

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

Web Services

- [Application Storage Service](#)
- [GeoIP Location Service](#)
-

Text Messaging Service

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

Electronic Mail

- [Domain Name Services Protocol](#)
- [Internet Message Access Protocol](#)
- [Mail Message Library](#)
- [Post Office Protocol](#)
- [Simple Mail Transfer Protocol](#)

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

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

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

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

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

Terminal Sessions

- [Rlogin Protocol](#)
- [Secure Shell Protocol](#)

- [Telnet Protocol](#)
- [Terminal Emulation](#)

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

Newsgroups

- [File Encoding Library](#)
- [Mail Message Library](#)
- [Network News Transfer Protocol](#)

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

General Concepts

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

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

Application Protocols

Throughout the documentation, you will see the word "protocols" mentioned. There are two general types of protocols that will be discussed in this developer's guide. The first type of protocol will be referred to as networking protocols. They are lower level protocols which define how data is exchanged between two systems. The two networking protocols that will be discussed are the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP).

Then there are what we will call the application protocols, which use the networking protocols to communicate. Application protocols deal with a specific type of functionality. For example, the File Transfer Protocol (FTP) is used to upload and download files, while the Simple Mail Transfer Protocol (SMTP) is used to send email messages. Conceptually, you can think of the networking protocols as defining the rules for how programs can communicate with one another over the Internet. The application protocols operate at a higher level, defining the rules for how a specific kind of task can be carried out, such as transferring a file from one computer to another.

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

RFC	Description
792	Internet Control Message Protocol
822	Standard for the Format of ARPA Internet Text Messages
854	Telnet Protocol Specification
868	Time Protocol
954	Nickname/Whois Protocol
959	File Transfer Protocol (FTP)
977	Network News Transfer Protocol
1034	Domain Name Services
1055	Serial Line IP (SLIP)
1282	Rlogin
1288	Finger User Information Protocol
1579	Firewall-Friendly FTP
1661	The Point-to-Point Protocol (PPP)
1738	Uniform Resource Locators
1869	SMTP Service Extensions
1939	Post Office Protocol Version 3
1945	Hypertext Transfer Protocol 1.0
1951	Deflate Compressed Data Format Specification
2045	Multipurpose Internet Mail Extensions (Part One)
2046	Multipurpose Internet Mail Extensions (Part Two)

2047	Multipurpose Internet Mail Extensions (Part Three)
2048	Multipurpose Internet Mail Extensions (Part Three)
2228	FTP Security Extensions
2616	Hypertext Transfer Protocol 1.1
2821	Simple Mail Transfer Protocol (SMTP)
2980	Common NNTP Extensions
3501	Internet Message Access Protocol Version 4

Transmission Control Protocol

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

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

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

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

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

TCP is known as a connection-oriented protocol. In other words, before two programs can begin to exchange data they must establish a connection with each other. This is done with a three-way handshake in which both sides exchange packets and establishes the initial packet sequence numbers. The sequence number is important because, as mentioned above, datagrams can arrive out of order; this number is used

to ensure that data is received in the order that it was sent. When establishing a connection, one program must assume the role of the client, and the other the server. The client is responsible for initiating the connection, while the server's responsibility is to wait, listen and respond to incoming connections. Once the connection has been established, both sides may send and receive data until the connection is terminated.

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

User Datagram Protocol

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

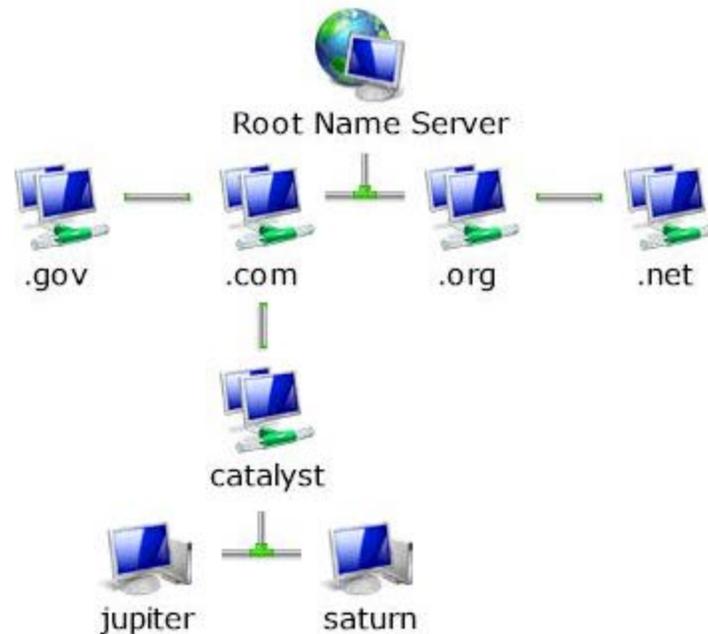
UDP is sometimes referred to as an unreliable protocol because when a program sends a UDP datagram over the network, there is no way for it to know that it actually arrived at its destination. This means that the sender and receiver must typically implement their own application protocol on top of UDP.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

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

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

Domain Names

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



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

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

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

Service Ports

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

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

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

Sockets

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

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

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

SocketTools Handles

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

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

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

Security Protocols

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

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

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

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

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

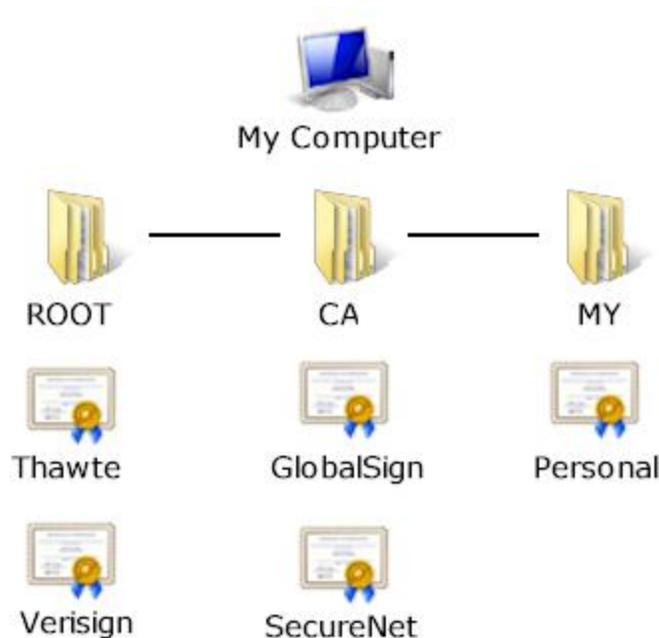
After the handshake phase has completed, the client may choose to examine the digital certificate that has been returned by the server. The information contained in the certificate includes the date that it was issued, the date that it expires, information about the organization who issued the certificate (called the issuer) and to whom the certificate was issued (called the subject of the certificate). The client may also validate the status of the certificate, determining if it was issued by a trusted certificate authority and was returned by the same company or individual it was issued to. There may be certain cases where the client determines that there's a problem with the certificate (for example, if the certificate's common name does

not match the domain name of the server), but chooses to continue communicating with the server. Note that the connection with the server will still be secure in this case. In other cases, for example if the certificate has expired, the client may choose to terminate the connection and warn the user.

Digital Certificates

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

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



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

system registry. Inadvertently removing a certificate could result in errors when attempting to access secure systems.

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

SocketTools Development

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

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

Protocols

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

Connections

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

Sessions

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

Authentication

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

Events

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

Application Design

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

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

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

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

Program Structure

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

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

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

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

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

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

Library Initialization

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

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

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

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

The runtime license key is normally stored in the Include folder where you installed SocketTools and is defined in a file named "csrtkey10" which can be included with your application. For example, C/C++ programmers would use the csrtkey10.h header file while Visual Basic programmers would use the csrtkey10.bas module. The C++ header file would look something like this:

```
//
// SocketTools 10
// Copyright 2023 Catalyst Development Corporation
// All rights reserved
//
// This file is licensed to you pursuant to the terms of the
// product license agreement included with the original software
// and is protected by copyright law and international treaties.

#ifndef _INCLUDE_CSRTKEY8_H
#define _INCLUDE_CSRTKEY8_H

#ifndef UNICODE
#define CSTOOLS10_LICENSE_KEY ((LPCSTR)NULL)
#else
#define CSTOOLS10_LICENSE_KEY ((LPCWSTR)NULL)
#endif

#endif /* _INCLUDE_CSRTKEY8_H */
```

The macro CSTOOLS10_LICENSE_KEY specifies your runtime license key and this is what you would pass as the first argument to the initialization function. For example, here is how a C program would initialize the File Transfer Protocol library:

```
BOOL bInitialized;
bInitialized = FtpInitialize(CSTOOLS10_LICENSE_KEY, NULL);
```

```

if (!bInitialized)
{
    TCHAR szError[256];
    DWORD dwError;
    dwError = FtpGetLastError();
    FtpGetErrorString(dwError, szError, 256);
    MessageBox(NULL, szError, NULL, MB_OK|MB_TASKMODAL);
    return;
}

```

Note that in this case, the key is defined as NULL, which means that a runtime license key has not been created yet. The `FtpInitialize` function would succeed when the program was executed on the development system where the evaluation copy of `SocketTools` was installed, but would fail on another system. When a license is purchased and the runtime license key is generated, the `csrtkey10.h` header file would be replaced with the actual key value. After re-compiling your program, you would be able to redistribute the application to other systems.

C++ programmers who are using the class wrappers are not required to explicitly initialize the libraries because this is done automatically in the class constructor. Review the `cstools10.h` file which defines all of the `SocketTools` classes to see how the constructor initializes the class.

For another example of how you can initialize the libraries, consider the Visual Basic module `csrtkey10.bas` which defines the runtime license key:

```

'
' SocketTools 10.0
' Copyright 2023 Catalyst Development Corporation
' All rights reserved
'
' This file is licensed to you pursuant to the terms of the
' product license agreement included with the original software
' and is protected by copyright law and international treaties.
'
Public Const CSTOOLS10_LICENSE_KEY As String = ""

```

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

```

Dim bInitialized As Long
bInitialized = FtpInitialize(CSTOOLS10_LICENSE_KEY, 0)

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

```

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

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

It is safe to call the initialization function more than once, but for each time that it is called, you must call the *Uninitialize* function for that library before your program terminates. For example, if you called **FtpInitialize** at the beginning of your program, you must call **FtpUninitialize** before your program ends. The *Uninitialize* function performs any necessary housekeeping operations, such as returning memory allocated for the library back to the operating system. If there are any open connections at the time that the Uninitialize function is called, they will be aborted. After the library has been uninitialized, you must call the initialization function again in order to use any of the library's other functions.

Synchronous and Asynchronous Sockets

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

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

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

Synchronous Sockets

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

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

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

Asynchronous Sockets

SocketTools facilitates the use of asynchronous sockets by generating events when appropriate. For

example, an event occurs whenever the server writes on the socket, which tells your application that there is data waiting to be read.

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

Best Practices

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

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

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

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

between blocking and non-blocking modes "on the fly" provides a powerful and convenient way to perform socket operations under some circumstances. However, switching between blocking and non-blocking mode can make the application more complex and difficult to debug. It is important to note that the warning regarding blocking sockets also applies here.

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

Secure Connections

The SocketTools Library Edition supports the ability to create secure connections using the standard SSH, SSL and TLS protocols. For those protocols which support them, enabling a secure connection is typically as simple as specifying an additional option when the Connect function is called. In some cases, certain protocols have additional options that control how the secure session is established.

Secure SSL/TLS connections may either be implicit or explicit, depending on the protocol. An implicit connection is one where the client and server begin negotiating the security options as soon as the connection is established. In most cases, a server which accepts secure implicit connections listens on a port number that is different from the default port it uses for standard, non-secure connections. An example of this is the Hypertext Transfer Protocol (HTTP) which accepts standard connections on port 80 and secure connections on port 443. When a client connects to port 443, the server automatically assumes that the client wants a secure connection.

On the other hand, an explicit connection requires that the client explicitly specify to the server that it wants a secure connection. Typically this is done by the client sending a command to the server that causes the server to begin negotiating with the client to establish a secure session. An example of this is the File Transfer Protocol, where the client can use the AUTH command to tell the server that it wants a secure connection. Servers may also support both explicit and implicit secure connections, based on which port the client connects to. SocketTools supports both implicit and explicit secure connections, and this is also controlled by the options specified to the library's Connect function.

In addition to establishing a secure connection, the client may be required to provide additional authentication information to the server in form a client certificate. A secure server may require that the client provide a certificate in addition to or instead of a username and password. To support this, your application must create the security credentials for the client and pass those to the Connect function. For more information, refer to the **SECURITYCREDENTIALS** structure in the Technical Reference. Additional information about secure connections and certificates is also available in the Developer's Guide.

Network Input/Output

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

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

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

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

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

1. Exchange the data as fixed length structures. This is the simplest approach, and has very little or no overhead. The client and server can either use predefined values, or negotiate the size of the data structures when the connection is established.
2. Prefix variable-length data structures with the number of bytes being sent. The length value could be expressed either as a native integer value, or as a fixed-length string that is converted to a numeric value by the application. This allows the receiver to read this fixed length value, and then use that value to determine how many additional bytes must be read to obtain the complete message or data structure.
3. Prefix the data with a unique byte or byte sequence that would normally not be found in the data stream. This would be followed by the data itself, with a complete message received when another unique byte sequence is encountered. Alternatively, a unique byte sequence could be used to terminate a message. This is the approach that many Internet

application protocols use, such as FTP, SMTP and POP3. Commands are sent as one or more printable characters, terminated with a carriage-return (CR) and linefeed (LF) byte sequence that tells the server that a complete command has been received.

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

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

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

Event Handling

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

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

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

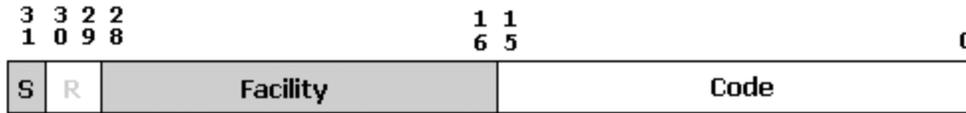
In addition to posting event notifications to a window, it is also possible to register an event handler that will be called by the library whenever the event occurs. In this case, you would use the **RegisterEvent** function for the library, where you would specify the event that you wish to register,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

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

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

Error Handling

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



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

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

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

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

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

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

Debugging Facilities

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

To provide logging functionality for your application, you must redistribute the `cstrcv10.dll` library along with those SocketTools libraries you are using in your program. The **`cstrcv10.dll`** library is what performs the actual logging and must be in a directory where it can be loaded by your application. It is recommended that you either install it in the Windows system directory or the directory where your application is installed. Note that this is a standard Windows dynamic link library and it does not need to be registered.

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

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

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

Language Support

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

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

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

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

Data Types

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

Description	Size	Range	C / C++	VB 6	VB.NET
Byte	1 byte	0 to 255	BYTE	Byte	Byte
Boolean	4 bytes	0 is False, 1 is True	BOOL	Long	Integer
Handle	4 bytes	0 to 4,294,967,295	HCLIENT	Long	Integer
Integer	4 bytes	-2,147,483,648 to 2,147,483,647	INT	Long	Integer
Integer	4 bytes	0 to 4,294,967,295	UINT	Long	Integer
Short Integer	2 bytes	-32,768 to 32,767	SHORT	Integer	Short
Short Integer	2 bytes	0 to 65,535	WORD	Integer	Short
Long Integer	4 bytes	-2,147,483,648 to 2,147,483,647	LONG	Long	Integer
Long Integer	4 bytes	0 to 4,294,967,295	DWORD	Long	Integer

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

Boolean Data

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

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

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

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

```
Dim bConnected As Long
bConnected = InetIsConnected(hSocket)
If bConnected <> 0 Then
```

```
    ' The socket is connected to a server
Else
    ' The socket is not connected, or the socket handle
    ' may be invalid; call InetGetLastError to check the
    ' error code
End If
```

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

String Data

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

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

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

Unicode

Unicode is a multi-language character set designed to encompass virtually all of the characters used with computers today. Unicode characters are represented by a 16-bit value on Windows, and differ from other character sets in two important ways. First, unlike the traditional single-byte (ANSI) character sets, Unicode is capable of representing significantly more characters in a variety of languages. Second, unlike multi-byte character sets (where some characters may be one byte in length, while others may be two, three or four bytes), the characters are fixed-width, which makes them easier to work with.

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

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

Automatic Encoding

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

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

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

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

Strings and Byte Arrays

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

C/C++ programmer may try to write code such as this:

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

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

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

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

#ifdef UNICODE
lpBuffer = (LPBYTE)_alloca((cchData + 1) * 4);

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

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

if (cchData <= 0)
{
    // Unable to convert the Unicode string
    return;
}
#else
pBuffer = (LPBYTE)lpszData;
#endif

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

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

Microsoft Visual C++

The SocketTools Library Edition includes import libraries, header files and class wrappers which can be used with Microsoft Visual C++ 6.0 and later. If you are using Visual C++ 6.0, it is required that you install Service Pack 6 (SP6). If you are using Visual C++ 7.1, which is part of Visual Studio .NET 2003, it is required that you install Service Pack 1 (SP1).

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

If you are programming in C or C++, you must include the header file **cstools10.h** in your program. This file contains all of the macro, type, structure and function prototype definitions used by the SocketTools libraries. This will also automatically include two other header files which are used by SocketTools. The **cserror10.h** header file defines all the error codes, and the **csrtkey10.h** header defines the runtime license key which is used to initialize the SocketTools libraries.

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

For Visual C++ programmers, there are special macros which are used in the cstools10.h header file that controls certain behavior:

CSTOOLS_NO_LIBRARIES

Visual C++ supports a pragma which can be used to automatically specify the names of libraries that the compiler should attempt to link with in order to resolve function calls. By default, it is not necessary to explicitly specify the names of the SocketTools import libraries to link to. However, if you wish to explicitly link to specific import libraries, then define this macro prior to including the cstools10.h header file.

CSTOOLS_NO_NAMESPACE

When compiling a C++ application, the SocketTools functions are defined in a namespace called SocketTools. This prevents the possibility of conflicts with functions of the same name that may be used in other libraries. If you prefer the functions to be defined in the global namespace instead, then define this macro prior to including the cstools10.h header file.

CSTOOLS_NO_CLASSES

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

These macros are not normally defined, and in most cases they will not be needed. Please note that if for some reason you modify the header files directly, you should copy the changes to another file using a different name. Otherwise, when you install an update to the product, the **cstools10.h** and **cserror10.h** header files will be replaced and you will lose your modifications.

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

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

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

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

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

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

Global Memory Handles

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

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

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

```
HGLOBAL hgb1Buffer = NULL;
DWORD dwLength = 0;
INT nResult;

nResult = HttpGetData(hClient,
    "/index.html",
    &hgb1Buffer,
    &dwLength,
    0);
```

```

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

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

    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}

```

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

The equivalent code in C# would look like this:

```

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

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

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

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

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

```

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

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

Event Handlers

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

seem confusing at first, it is easiest to think of them as function pointers. They're basically used in the same way that function pointers are used in C++, except that they're type-safe.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Microsoft Visual Basic .NET

The SocketTools Library Edition provides a Visual Basic module in the Include folder named `cstools10.vb` which can be included with your projects. This defines the constants and functions in the SocketTools libraries.

String Arguments

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

```
Dim strLocalName As String
Dim nLength As Integer

nLength = InetGetLocalName(strLocalName, 256)
```

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

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

nLength = InetGetLocalName(strLocalName, 256)
```

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

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

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

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

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

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

Byte Array Arguments

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

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

```

INT nResult;

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

```

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

```

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

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

```

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

```

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

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

```

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

Global Memory Handles

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

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

The solution for this problem is to use a global memory handle rather than a pre-allocated buffer. Instead of copying the data into a buffer, the function allocates a global memory handle and stores the contents in the memory referenced by that handle. When the function returns, it passes the handle back to the caller.

The caller then dereferences the handle to access the memory, and releases the handle when it is no longer needed. Here is an example of how it would be used in C/C++:

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

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

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

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

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

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

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

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

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

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

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

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

    ' Do something with the data
End If
```

If you wanted to convert the contents of the global memory buffer into a string, the Marshal class has a helper function which enables you to do this easily. To use it, you should import

System.Runtime.InteropServices and then use the **PtrToStringAnsi** method. For example:

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

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

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

    ' Do something with the data
End If
```

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

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

Event Handlers

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

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

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

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

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

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

```

Private Sub HttpEventHandler(ByVal hClient As IntPtr, _
                           ByVal nEventId As Integer, _
                           ByVal dwError As Integer, _
                           ByVal dwParam As IntPtr)
    Select Case nEventId
    Case HTTP_EVENT_PROGRESS
        Dim httpStatus As HTTPTRANSFERSTATUS
        HttpGetTransferStatus(hClient, httpStatus)
        ProgressBar1.Value = CInt(100.0# * _
            CDb1(httpStatus.dwBytesCopied) / _
            CDb1(httpStatus.dwBytesTotal))
    End Select
End Sub

```

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

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

```

Dim httpEventProc As HttpEventDelegate = New HttpEventDelegate(AddressOf
    HttpEventHandler)

```

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

```

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

```

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

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

Microsoft Visual Basic 6.0

The SocketTools Library Edition provides a module for Visual Basic 6.0 in the Include folder named `cstools10.bas` which can be included with your projects. This defines the constants and functions in the SocketTools libraries. It is recommended that you install at least Visual Studio 6.0 Service Pack 5 (SP5) for Visual Basic 6.0.

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

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

Library Name	Library Description
<code>csdnsv10.dll</code>	Domain Name Service
<code>csftpv10.dll</code>	File Transfer Protocol
<code>cshttpv10.dll</code>	Hypertext Transfer Protocol
<code>csicmv10.dll</code>	Internet Control Message Protocol
<code>csmapv10.dll</code>	Internet Message Access Protocol
<code>csmsgv10.dll</code>	Mail Message
<code>csmtpv10.dll</code>	Simple Mail Transfer Protocol
<code>csncdv10.dll</code>	File Encoding Library
<code>csnvtv10.dll</code>	Terminal Emulation
<code>csnwsv10.dll</code>	Network News Transfer Protocol
<code>cspopv10.dll</code>	Post Office Protocol
<code>csrshv10.dll</code>	Remote Command Protocol
<code>csrsv10.dll</code>	News Feed Library
<code>cstimv10.dll</code>	Time Protocol
<code>cstntv10.dll</code>	Telnet Protocol
<code>cstxtv10.dll</code>	Text Message Library
<code>cswhov10.dll</code>	Whois Protocol
<code>cswskv10.dll</code>	SocketWrench (Windows Sockets)

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

String Arguments

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

```
Dim strLocalName As String
Dim nLength As Long

nLength = InetGetLocalName(strLocalName, 256)
```

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

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

nLength = InetGetLocalName(strLocalName, 256)
```

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

```
Dim strLocalName As String
Dim nLength As Long

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

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

```
Dim strLocalName As String
Dim nLength As Long

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

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

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

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

Byte Array Arguments

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

```

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

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

```

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

```

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

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

```

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

```

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

```

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

Global Memory Handles

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

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

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

```

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

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

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

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

    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}

```

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

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

```

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

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

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

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

```

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

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

```

Declare Function HttpGetData Lib "cshtpv10.dll" _

```

```

Alias "HttpGetDataA" ( _
ByVal hClient As Long, _
ByVal lpszResource As String, _
ByRef lpBuffer As Byte, _
ByRef lpdwLength As Long, _
ByVal dwOptions As Long _
) As Long

```

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

```

Declare Function HttpGetData Lib "cshttpv10.dll" _
Alias "HttpGetDataA" ( _
ByVal hClient As Long, _
ByVal lpszResource As String, _
ByRef hgblBuffer As Long, _
ByRef lpdwLength As Long, _
ByVal dwOptions As Long _
) As Long

```

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

```

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

hgblBuffer = 0
dwLength = 0

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

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

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

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

    GlobalUnlock hgblBuffer
    GlobalFree hgblBuffer
End If

```

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

Event Handlers

SocketTools uses events to notify the application when some change in status occurs, such as when data is available to be read or progress notifications during a file transfer. An event handler is simply a callback function which has a specific set of arguments, and the address of that function is passed to the **HttpRegisterEvent** function in the library. In Visual Basic 6.0, you can use the **AddressOf** operator to obtain the address of a public callback function to pass to the function.

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

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

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

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

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

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

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

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

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

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

Your event handler is now registered, and SocketTools will call your event handler during the process of

downloading or uploading a file to notify you of the progress of the transfer.

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

Borland Delphi

The SocketTools Library Edition provides a Delphi unit in the Include folder named `cstools10.pas` which can be included with your projects. This defines the constants and functions in the SocketTools libraries.

String Arguments

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

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

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

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

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

Early versions of Delphi had helper functions such as **StrNew** to help convert and allocate null terminated strings. However, those functions have been deprecated and in most cases current versions of Delphi will automatically convert between character arrays and its native **String** type. There are also a number of functions available which are designed specifically to work with null terminated strings. Refer to your language reference for more information about how Delphi uses null terminated strings.

Byte Array Arguments

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

```

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

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

```

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

```

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

```

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

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

```

strBuffer := PAnsiChar(@byteBuffer);

```

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

```

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

```

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

Global Memory Handles

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

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

the size of a resource on a web server. Another alternative would be to simply allocate a very large buffer, but this could result in the application allocating large amounts of memory that it doesn't use and you would still run the risk that it wouldn't be large enough.

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

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

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

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

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

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

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

The equivalent code in Delphi would look like this:

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

    nResult := HttpGetData(hClient,
                          PChar('/index.html'),
                          hgblBuffer,
                          dwLength,
                          0);

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

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

```

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

```

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

Event Handlers

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

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

```

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

```

In Delphi, the equivalent function would be defined as:

```

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

```

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

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

```

procedure HttpEventHandler(
    hClient: Integer;
    nEventId: Integer;
    dwError: LongWord;
    dwParam: LongWord); stdcall;
var
    httpStatus: HTTPTRANSFERSTATUS;
    nPercent: Integer;
begin
    if nEventId = HTTP_EVENT_PROGRESS then
    begin
        HttpGetTransferStatus(hClient, httpStatus);
        nPercent := Trunc(100.0 * httpStatus.dwBytesCopied /
                        httpStatus.dwBytesTotal);
        Form1.ProgressBar1.Position := nPercent;
    end;
end;

```

This event handler checks to see if the event ID indicates that it is a progress event, and if it is, calls

HttpGetTransferStatus to determine the number of bytes that have been copied so far. This is used to calculate a percentage and a progress bar control is updated with that value.

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

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

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

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

PowerBASIC

The SocketTools Library Edition provides a complete set of function declarations and constants in the Include folder named `cstools10.inc` which can be included with your projects. It is recommended that you use version 10.0 or later of the PowerBASIC compiler. The samples included with SocketTools were developed using version 10.0 of the compiler and version 2.0 of the form designer.

String Arguments

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

```
VOID ShowError()  
{  
    TCHAR szError[256];  
    DWORD dwError;  
  
    dwError = HttpGetLastError();  
    if (dwError > 0)  
    {  
        HttpGetErrorString(dwError, szError, 256);  
        MessageBox(NULL, szError, "Error", MB_OK);  
    }  
}
```

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

```
SUB ShowError()  
    LOCAL szError AS STRINGZ * 256  
    LOCAL dwError AS DWORD  
  
    dwError = HttpGetLastError()  
    IF dwError <> 0 THEN  
        HttpGetErrorString(dwError, szError, SIZEOF(szError))  
        MSGBOX szError, %MB_OK OR %MB_ICONEXCLAMATION, "Error"  
    END IF  
END SUB
```

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

Byte Array Arguments

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

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

```

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

```

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

```

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

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

```

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

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

```

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

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

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

```

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

Global Memory Handles

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

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

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

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

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

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

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

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

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

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

The equivalent code in PowerBASIC would look like this:

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

hgblBuffer = %NULL
dwLength = 0

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

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

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

```

        GlobalUnlock(hgblBuffer)
        GlobalFree(hgblBuffer)
    END IF

```

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

Event Handlers

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

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

```

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

```

In PowerBASIC, the equivalent function would be defined as:

```

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

```

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

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

```

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

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

    hDlg = dwParam ' Dialog with the progress bar control

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

```

END SUB

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

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

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

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

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

SocketTools Library Overview

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

Application Storage

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

Domain Name Service

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

Encoding, Compression and Encryption

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

File Transfer Protocol

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

GeoIP Location

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

Hypertext Transfer Protocol

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

Internet Control Message Protocol

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

[Internet Message Access Protocol](#)

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

[Mail Messages](#)

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

[Network News Transfer Protocol](#)

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

[News Feeds](#)

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

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

Post Office Protocol

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

Remote Commands

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

Secure Shell Protocol

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

Simple Mail Transfer Protocol

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

SocketWrenchy

The SocketWrench library provides a higher-level interface to the Windows Sockets API, designed

to be suitable for programming languages other than C and C++. If needed, function calls can be intermixed between the SocketWrench and Windows Sockets libraries. In addition, the SocketWrench supports secure communications using Transport Layer Security (TLS).

Telnet Protocol

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

Terminal Emulation

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

Text Messaging

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

Time Protocol

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

Whois Protocol

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

Application Storage Service

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

The connection to the storage service is always secure, using TLS 1.2 and AES-256 bit encryption. There are no third-party services you need to subscribe to, and there are no additional usernames or passwords for you to manage. Access to the service is associated with an account which is created when you purchase a development license, and the security tokens are bound to the runtime license key used when initializing the API. You also have the option to compress and encrypt your data you store using the [Encoding, Compression and Encryption](#) APIs.

Terminology

When you get started with the storage API, you'll notice there is some different terminology which is used. This will provide an overview of that terminology, and compare it to common terms used with traditional protocols like FTP. When accessing an FTP server, you generally deal with *directories*, *files*, *names* and *types* (generally whether the file is binary or text). The storage API has similar concepts, but uses somewhat different terminology.

Application Identifiers

An application identifier (AppId) is a null terminated string which uniquely identifies your application. This string, used in conjunction with your runtime license key, is used to generate an access token. This token is used to access the storage container which contains the data which you've stored.

It is recommended you use a standard format for the AppId which consists of your company name, application name and optionally a version number. Some examples of an AppId string would be:

- MyCompany.MyApplication
- MyCompany.MyApplication.1

It is important to note with these two example IDs, although they are similar, they reference two different applications. Objects stored using the first ID will not be accessible using the second ID. If you want to store objects which should be shared between all versions of the application, it is recommended you use the first form, without the version number. If you want to store objects which should only be accessible to a specific version of your application, then it is recommended you use the second form which includes the version number.

The AppId must only consist of ASCII letters, numbers, the period and underscore character. Whitespace characters and non-ASCII Unicode characters are not permitted. The maximum length of the string is 64 characters, including the terminating null character. It is not required for your application to create a unique AppId. Each storage account has a default internal AppId named **SocketTools.Storage.Default**. This AppId is used if a NULL pointer or an empty string is specified.

Containers

Storage containers are somewhat analogous to directories or folders in a file system, however they are general purpose and designed to allow you to control how your application accesses the data

that's been stored. There are four container types which are defined by the API, and you can think of them as types of boxes or file cabinets which you store your data in.

It is important to keep in mind these containers are available to all users of your application, your program controls who has access to any particular data file. Your users will not be able to "browse" any of the containers unless you specifically provide that capability by implementing it in your own code. There is no public access to any of the data which you upload, and our service does not use an open API accessible by third parties.

WEB_STORAGE_GLOBAL

The global storage container which is available to all users of your application. Any data stored in this container is available to everyone who uses your software. Unless you have a specific need to limit access to the data to a specific user or group of users, this is the recommended container you use to store data.

WEB_STORAGE_DOMAIN

The domain storage container is limited to users in the same local domain, defined either by the name of the domain or workgroup assigned to the computer system. This can provide a kind of organization wide storage, but it does depend on the domain being unique. For example, if you are using domain storage for your application, and you have multiple customers who have systems part of the default "Workgroup" domain, they would all share the same container. If the domain or workgroup name changes, then data stored in the container would no longer be available.

WEB_STORAGE_MACHINE

The local machine storage container is associated with the physical computer system your application is running on. The machine is identified by unique characteristics of the system, including the boot volume GUID. Data stored in this container can only be accessed from the application running on that particular system. If the operating system is reinstalled, the machine ID will change and data stored in this container would no longer be available.

WEB_STORAGE_USER

The current user storage container is associated with the current user who is using your application. The user identifier is based on the Windows Security Identifier (SID) assigned to the account when it's created. If the user account is deleted, the data stored in this container will no longer be available to the application. Another user on the same computer system would not be able to access the data in this container.

If you decide to use anything other than global storage, the data your application stores can be orphaned if the system configuration or user account changes. It's recommended you store critical application data and general configuration information using **WEB_STORAGE_GLOBAL** and use other non-global storage containers for configuration information which is unique to that system and/or user which is not critical and can be easily recreated. If you're concerned about protecting the data you upload to global storage, you can encrypt it prior to storing it.

Objects

Storage objects are similar to files in a file system. They are discrete blocks of data, associated with a label (name), have attributes and are associated with a particular content type. However, an object does not need to be an actual file on the local system. For example, you could store an object which is a string, a pointer to a structure, or any block of memory. You could also just store a complete file as an object. Unlike files, you cannot perform partial reads of an object or "seek" into certain parts of a stored object. Of course, you can download an object, either in memory or to a local file, and perform whatever operations you require on the data.

Labels

Object labels are similar to file names, and are a way to identify a stored object instead of using its internal object ID. However, there are some important differences. The most significant difference being labels are case-sensitive, unlike Windows file names. An object with the label "AppConfig" is considered to be different than one with the label "appconfig". Labels can contain Unicode characters, but they cannot contain control characters.

You can also use forward slashes or backslash characters in the label, but it's important to note objects are not stored in a hierarchical structure. Your application can store objects using a folder-like structure, but it's not something which is enforced by the API.

Media Type

Each object your application stores is associated with a media type (also called a content type) which identifies the object's data. This uses the standard MIME media type designations, such as "text/plain" or "application/octet-stream". Your application can explicitly specify the media type you want to associate with the object, or you can have the API choose for you, based on the contents of the object and using the label as a hint for what it may contain. For example, if you create an object with the label "AppConfig.xml" and it contains text, then the API will select "text/xml" as the default media type.

Initialization

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

[WebInitialize](#)

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

[WebOpenStorage](#)

Opens a storage container for your application and returns a handle which is used with other functions.

[WebCloseStorage](#)

Close the storage container and release the resources allocated for the session.

[WebUninitialize](#)

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

Data Storage

The library provides functions to upload and download to the storage container. You can store local files, or you can create objects from data in memory. There are also functions which make it easier to store and retrieve textual data using null terminated strings.

[WebGetFile](#)

Download the object data and store in a file on the local system. There is also an extended version of the function named [WebGetFileEx](#) which provides additional options, such as the ability to retrieve information about the object downloaded in a single function call.

[WebGetObject](#)

Download the object data and store it in a memory buffer provided by the caller. The buffer can either be a pre-allocated block of memory, or reference a global memory handle which will contain the data when the function returns.

[WebGetTextObject](#)

Download the object data and store it in a null terminated string. This function would be used with data which is only textual, and it may contain Unicode text. If the the Unicode version of the function is called, the text is automatically converted from UTF-8 encoded to UTF-16 encoded text.

[WebPutFile](#)

Upload the the contents of a local file and store it as an object in the current container. There is also an extended version of the function named [WebPutFileEx](#) which provides additional options, such specifying the object attributes, media type and information about the stored object when the function returns.

[WebPutObject](#)

Upload the object data from a buffer in memory and store it as an object in the current container. This function would typically be used to store binary data, including compressed or encrypted text.

[WebPutTextObject](#)

Upload the contents of a string and store it as an object in the current container. The text is specified as a null terminated string, and if the Unicode version of the function is used, the text will be converted and stored as UTF-8 encoded text. The [WebGetTextObject](#) function will automatically convert the text back to UTF-16 encoding when the text object is retrieved from storage.

Data Management

The data management functions allow you to obtain information about stored objects and perform typical operations such as copying, renaming and deleting objects from the container.

[WebGetObjectInformation](#)

Returns information about a specific object stored in the container. This function populates a [WEB_STORAGE_OBJECT](#) structure which will contains metadata for the object such as its ID, size, attributes, creation and modification times.

[WebGetFirstObject](#)

Enables your application to search for and enumerate objects in a container based on their label and/or their media type. This function is used in conjunction with the [WebGetNextObject](#) function to list all matching objects in a container.

[WebCompareFile](#)

Compares the contents of a local file with the data in a stored object. This function can be used to determine if the contents of a file have changed since the file was previously stored using the [WebPutFile](#) function.

[WebCompareObject](#)

Compares the contents of data in memory with the data in a stored object. This function can be used to determine if the contents of the buffer have changed since the data was previously stored using the [WebPutObject](#) function.

[WebCompareText](#)

Compares the contents of null terminated string with the text in a stored object. This function can be used to determine if the contents of the string have changed since the data was previously stored using the [WebPutTextObject](#) function. If the Unicode version of this function is used, the string is automatically converted to use UTF-8 encoding prior to being compared with the contents of the stored object.

[WebCopyObject](#)

Copies the contents of a stored object to a new container, or duplicating the object within the same container using a different label. Information about the newly created object is returned to the caller.

[WebMoveObject](#)

Moves the contents of a stored object to a new container. Information about the object which has been moved will be returned to the caller.

[WebRenameObject](#)

Changes the label associated with a stored object. The new label for the object cannot already exist in the same container. If you want to change the label to one already assigned to an existing object, the object must first be deleted.

[WebDeleteObject](#)

Removes the stored object from the container. This operation is immediate and permanent. Deleted objects cannot be recovered by the application at a later time.

[WebGetStorageQuota](#)

Returns information about how much storage space is available. It will populate a `WEB_STORAGE_QUOTA` structure which will specify how many bytes of storage you're using and how many objects you have created. It is important to note accounts created with an evaluation license have much lower quota limits than a standard account and should be used for testing purposes only. After the evaluation period has ended, all objects stored using the evaluation license will be deleted.

[WebResetStorage](#)

Resets the storage container and deletes all objects which were stored there. This function resets the container back to its initial state, deleting all object metadata from the database and removing all stored data. This operation is immediate and the objects stored in the container are permanently deleted. They cannot be recovered by your application.

Other Functions

Several additional utility functions are available as part of the storage API, including functions to register and de-register application identifiers and validate object labels.

[WebRegisterAppId](#)

Register a new application identifier (AppId) to be used to access a storage container. It is not required you create a unique application ID, but it can be helpful to distinguish stored content between different versions of your applications.

[WebUnregisterAppId](#)

Unregister an application identifier which was previously registered by your application. You should be extremely careful when using this function because it permanently delete all stored objects created using the AppId value. Internally it revokes the access token granted to your application and causes the server to expunge all objects in the container associated with the token.

[WebValidateAppId](#)

A utility function which can be used to validate an application identifier, ensuring it is valid and has been registered.

[WebValidateLabel](#)

A utility function which can be used to validate an object label to ensure it does not contain any invalid characters. This would be primarily used by applications which allow a user to specify the label names for the objects being stored.

Domain Name Service

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

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

[DnsInitialize](#)

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

[DnsCreateHandle](#)

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

[DnsCloseHandle](#)

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

[DnsRegisterServer](#)

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

[DnsUninitialize](#)

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

Address Conversion

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

[DnsGetAddress](#)

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

[DnsFormatAddress](#)

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

Host Tables

When resolving a host name or IP address, the library will first search the local system's host table, a file

that is used to map host names to addresses. On Windows 95/98 and Windows Me, if the file exists it is usually found in C:\Windows\hosts. On Windows NT and later versions, it is found in C:\Windows\system32\drivers\etc\hosts. Note that the file does not have an extension.

[DnsGetDefaultHostFile](#)

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

[DnsGetHostFile](#)

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

[DnsSetHostFile](#)

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

Host Name Resolution

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

[DnsGetHostAddress](#)

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

[DnsGetHostName](#)

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

Mail Exchange Records

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

[DnsEnumMailExchanges](#)

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

Advanced Queries

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

[DnsGetHostInfo](#)

Return additional information about the specified host name. If the name server has been configured to provide host information for the domain, this function will return that data. Typically

it is used to indicate what hardware and operating system the host uses.

[DnsGetHostServices](#)

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

[DnsGetRecord](#)

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

Encoding, Compression and Encryption

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

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

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

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

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

Encoding Types

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

Base64

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

Quoted-Printable

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

Uucode

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

yEnc

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

Data Encoding

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

[EncodeFile](#)

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

[DecodeFile](#)

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

[EncodeBuffer](#)

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

[DecodeBuffer](#)

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

[IsUnicodeText](#)

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

[UnicodeDecodeText](#)

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

[UnicodeEncodeText](#)

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

Data Compression

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

[CompressFile](#)

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

[ExpandFile](#)

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

[CompressBuffer](#)

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

[ExpandBuffer](#)

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

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

Data Encryption

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

[AesEncryptFile](#)

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

[AesDecryptFile](#)

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

[AesEncryptBuffer](#)

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

[AesDecryptBuffer](#)

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

[AesEncryptString](#)

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

[AesDecryptString](#)

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

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

File Transfer Protocol

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

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

FtpInitialize

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

FtpConnect

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

FtpProxyConnect

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

FtpLogin

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

FtpDisconnect

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

FtpUninitialize

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

File Transfers

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

FtpGetData

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

[FtpGetFile](#)

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

[FtpGetMultipleFiles](#)

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

[FtpPutData](#)

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

[FtpPutFile](#)

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

[FtpPutMultipleFiles](#)

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

File Management

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

[FtpDeleteFile](#)

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

[FtpRenameFile](#)

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

[FtpGetFileStatus](#)

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

[FtpGetFileSize](#)

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

[FtpGetFileTime](#)

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

[FtpSetFileTime](#)

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

[FtpGetFilePermissions](#)

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

[FtpSetFilePermissions](#)

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

Directory Management

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

[FtpOpenDirectory](#)

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

[FtpGetFirstFile](#)

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

[FtpGetNextFile](#)

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

[FtpChangeDirectory](#)

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

[FtpCreateDirectory](#)

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

[FtpRemoveDirectory](#)

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

GeoIP Location Service

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

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

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

Functions

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

[WebInitialize](#)

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

[WebGetLocation](#)

This function populates a [WEB_LOCATION](#) structure which contains information about the physical location of the system. This structure has a number of members which provide details about the location:

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

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

	These names are defined by the Internet Assigned Numbers Authority (IANA) and have values like "America/Los_Angeles" and "Europe/London".
szTzShortName	A string which specifies the abbreviated time zone code. If daylight savings time is used within the time zone, then this value can change based on whether or not daylight savings is in effect. If a short time zone code cannot be determined, a value such as "UTC+9" may be returned, indicating the number of hours ahead or behind UTC.
nTimezoneOffset	A integer which specifies the number of seconds east or west of the prime meridian (UTC). A positive value indicates a time zone which is east of the prime meridian and a negative value indicates a time zone which is west of the prime meridian.
dLatitude	A real number which specifies the latitude of the location in decimal format. A positive value indicates a location which is north of the equator, while a negative value is a location which is south of the equator.
dLongitude	A real number which specifies the longitude of the location in decimal format. A positive value indicates a location which is east of the prime meridian, while a negative value is a location which is west of the prime meridian.
stLocalTime	A SYSTEMTIME structure which contains information about the current date and time at the location, adjusted for its time zone and whether or not it's in daylight savings time.

[WebUninitialize](#)

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

Hypertext Transfer Protocol

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

File Transfers

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

Script Execution

Another common use for HTTP is to execute scripts on the web server. The application can pass additional data to the script, which is similar in concept to how arguments are passed to a command that is entered from the command prompt. This uses the standard POST command, and the resulting output from the script is returned back to the application where it can be displayed or processed. An application can use the **HttpCommand** function to execute the script and then process the output in code, or can use the higher level function **HttpPostData** which will execute the script and return the output from that script in a single function call.

Uniform Resource Locators

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

```
http://[username : [password] @] remotehost [:remoteport] / resource [ ? parameters ]
```

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

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

The first step that your application must take is to initialize the library and then establish a connection. The

following functions are available for use by your application:

[HttpInitialize](#)

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

[HttpConnect](#)

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

[HttpProxyConnect](#)

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

[HttpAuthenticate](#)

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

[HttpDisconnect](#)

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

[HttpUninitialize](#)

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

File Transfers

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

[HttpGetData](#)

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

[HttpGetFile](#)

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

[HttpPutData](#)

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

[HttpPutFile](#)

This function uploads a file from the local system to the server using the PUT command. Not all servers support this command, and some may require that the client authenticate prior to calling

this function.

[HttpPostFile](#)

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

File Management

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

[HttpGetFileSize](#)

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

[HttpGetFileTime](#)

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

[HttpDeleteFile](#)

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

[HttpCommand](#)

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

Script Execution

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

[HttpGetData](#)

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

```
/cgi-bin/test.cgi?data1=value1&data2=value2
```

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

[HttpPostData](#)

An alternative method of providing information to a script is to post data to the script. Instead of the data being part of the resource name itself, posted data is sent separately and is provided as input to the script. This is the same method that is typically used when a user clicks the Submit

button on a web-based form. This function requires the name of the script and the address of a buffer that contains the data that will be posted. The resulting output from the script is returned to the caller in the same way that **HttpGetData** works.

Internet Control Message Protocol

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

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

[IcmpInitialize](#)

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

[IcmpCreateHandle](#)

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

[IcmpCloseHandle](#)

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

[IcmpUninitialize](#)

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

Ping and TraceRoute

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

[IcmpEcho](#)

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

[IcmpTraceRoute](#)

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

[IcmpSendEcho](#)

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

sent, and is typically used with asynchronous sessions.

[IcmpRecvEcho](#)

This is a lower level function that will receive a single ICMP echo reply datagram that was returned by the server. Typically this is used to receive datagrams that were sent in response to the **IcmpSendEcho** function. The **IcmpGetTripTime** function can be used after the function returns in order to check the amount of time it took to receive the datagram.

Internet Message Access Protocol

The Internet Message Access Protocol (IMAP) is an application protocol which is used to access a user's email messages which are stored on a mail server. However, unlike the Post Office Protocol (POP) where messages are downloaded and processed on the local system, the messages on an IMAP server are retained on the server and processed remotely. This is ideal for users who need access to a centralized store of messages or have limited bandwidth. The SocketTools IMAP library implements the current standard for this protocol, and provides functions to retrieve messages, create and manage mailboxes, and search for specific messages based on some user-defined search criteria.

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

[ImapInitialize](#)

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

[ImapConnect](#)

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

[ImapDisconnect](#)

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

[ImapLogin](#)

Authenticate yourself to the server using a username and password. This function should be called immediately after the connection has been established to the server.

[ImapUninitialize](#)

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

Managing Mailboxes

One of the primary differences between the IMAP and POP3 protocol is that IMAP is designed to manage messages on the mail server, rather than downloading all of the messages and storing them on the local system. To support this, IMAP allows the client to maintain multiple mailboxes on the server, which are similar in concept to message folders used by mail client software. A mailbox can contain messages, and in some cases a mailbox can contain other mailboxes, forming a hierarchy of mailboxes and messages, similar to directories and files in a filesystem. A special mailbox named INBOX contains new messages for the user, and additional mailboxes can be created, renamed and deleted as needed. Here are the most important functions for managing mailboxes:

[ImapCheckMailbox](#)

Check the mailbox for any new messages which may have arrived. Because messages are managed on the server, it is possible for new mail to arrive during the client session.

[ImapCreateMailbox](#)

Create a new mailbox on the server with the specified name.

[ImapDeleteMailbox](#)

Delete a mailbox from the server. Most servers will only permit a mailbox to be deleted if it does not contain any mailboxes itself. Unlike deleting a message, which can be undeleted, deleting a mailbox is permanent.

[ImapExamineMailbox](#)

Once the session has been established and authenticated, a mailbox should be selected. This enables the client to manage the messages in that mailbox. This function selects the specified mailbox in read-only mode so that messages can be read, but not modified. To select the mailbox in read-write mode, use the `ImapSelectMailbox` function.

[ImapExpungeMailbox](#)

Remove all of the messages marked for deletion and return updated status information about the current mailbox.

[ImapGetFirstMailbox](#)

This function begins the process of enumerating the available mailboxes on the server, according to certain criteria provided to the function.

[ImapGetNextMailbox](#)

This function returns the next mailbox from the server, based on the criteria specified to the **ImapGetFirstMailbox** function. When all the mailboxes have been returned, this function will return an error indicating that there are no more mailboxes.

[ImapRenameMailbox](#)

Renames an existing mailbox. One of the interesting uses of this function is the ability to rename the special INBOX mailbox. Instead of actually renaming it, it moves all of the messages to the new mailbox and empties the INBOX.

[ImapSelectMailbox](#)

Once the session has been established and authenticated, a mailbox should be selected. Selecting a mailbox enables the client to manage the messages in that mailbox. This function selects the specified mailbox in read-write mode so that changes can be made to the mailbox.

[ImapUnselectMailbox](#)

This function unselects the currently selected mailbox, and allows the caller to specify if messages marked for deletion should be expunged (removed) from the mailbox or reset back to an undeleted state.

Managing Messages

There are functions in the IMAP library for managing messages which enables the application to create, delete and move messages. To use these functions, a mailbox must be selected, either by calling the **ImapSelectMailbox** or **ImapExamineMailbox** function. Functions which modify the mailbox require that it be opened in read-write mode. Messages are identified by a number, starting with one for the first message in the mailbox.

[ImapCopyMessage](#)

Copy a message to a specific mailbox.

[ImapDeleteMessage](#)

Mark the specified message for deletion. Unlike the POP3 protocol, when a message is deleted on an IMAP server it can still be accessed. The message will not actually be removed from the mailbox unless the mailbox is expunged, unselected or the client disconnects from the server.

[ImapEnumMessages](#)

Return information about all, or a specific range of, messages on the server.

[ImapGetMessageCount](#)

Return the number of messages in the currently selected mailbox.

[ImapGetMessageFlags](#)

Return the status flags for a specific message. Messages have a number of flags which can be set that determines their status in the mailbox. For example, messages can be flagged as answered,

seen (read), recent (new) or deleted.

[ImapGetMessageSize](#)

Return the size of the message in bytes.

[ImapSetMessageFlags](#)

Set one or more flags for the specified message. Flags can be cleared, added or replaced using this function.

[ImapUndeleteMessage](#)

Remove the deletion flag from the specified message.

Viewing Messages

One of the more powerful features of the IMAP protocol is the ability to precisely select what kinds of message data you wish to retrieve from the server. It is possible to retrieve only specific headers, or specific sections of a multipart message. Because IMAP understands MIME formatted messages, it is possible to only retrieve the textual portion of a message without having to download any attachments that may have come with it.

[ImapGetHeaderValue](#)

This function returns the value for a specified header field in the message. Using this function, it is not necessary to download and parse the message header. To obtain the value of a header field in a specific part of a multipart message, use the **ImapGetHeaderValueEx** function.

[ImapOpenMessage](#)

This is a lower level function which opens a message for reading from the server. A more complex version of this function called **ImapOpenMessageEx** function allows you to specify the type of message data that you want, a specific part of a multipart message, and a byte offset into the message. The application would then call **ImapRead** to read the contents of the message, followed by **ImapCloseMessage** when all the message data has been read. Also see the **ImapGetMessage** function, which will return the contents of a message into a memory buffer.

[ImapGetMessageParts](#)

This function returns the number of parts in a multipart message and is useful for determining if a message is a simple message or a MIME formatted message with multiple parts, such as one that includes file attachments.

Downloading Messages

In some cases, it may be preferable to download a complete message from the server to the local system. This can be easily done with a single function call.

[ImapGetMessage](#)

This function retrieves the specified message and stores it in a buffer provided by the caller; you can specify the type of message data that you want, a specific part of a multipart message and the amount of data that you want. For example, it is possible to request that only the first 1500 bytes of the body of the 3rd part of a multipart message should be returned.

[ImapStoreMessage](#)

This function downloads a complete message and stores it as a text file on the local system.

Mail Message

The Mail Message library can be used to create and process messages in the format defined by the Multipurpose Internet Mail Extensions (MIME) standard. When a message is parsed, it is broken into parts, each consisting of two sections. The first part is called the header section and it describes the format of the data and how it should be represented to the user. The second section is the data itself. A typical mail message without file attachments has one part, with the body of the message being the data. Messages with attachments have multiple parts, each with a header describing the type of data. The library can be then used to extract the data from a multipart message and save it to a file on the local system, delete the part from the message, or add additional parts to the message, such as attaching a file.

The library can also be used to create new multipart messages with alternative content, such a message with both plain text and styled HTML text. Once a message has been created, files can be attached to the message and the application can make any other changes that are needed. The library provides complete access to all headers and content in a multipart message, including the ability to create your own custom headers and make modifications to specific sections.

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

[MimeInitialize](#)

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

[MimeComposeMessage](#)

Compose a new message using the specified header field values and content. Using this function, you can create a message with the From, To, Cc and Subject headers already defined, along with any text for the message. You can also optionally provide both plain and styled HTML text versions of the message and the function will automatically create a multipart message. The function returns a handle which can be used by the application to make further modifications, such as attaching files to the message.

[MimeCreateMessage](#)

Create a new, empty message and return a handle which can be used by the application to reference the message. In most cases, using the MimeComposeMessage function is preferred, however there may be some situations where the application needs a message handle when the contents of the message aren't known at the time. For example, parts of a message may be created dynamically based on the results of several database queries.

[MimeDeleteMessage](#)

Releases the memory allocated for the message and destroys the message handle. After this function returns, the handle is no longer valid. This function should be called when the message is no longer needed by the application.

[MimeUninitialize](#)

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

Message Headers

Each message has one or more headers fields which provide information about the contents of the message. For example, the "From" header field specifies the email address of the person who sent the message. There are a fairly large number of header fields defined by the MIME standard, and applications can also create their own custom headers if they wish. The library gives the application complete access to the header fields in a message. Headers can be examined, modified, created or removed from the

message as needed.

[MimeGetMessageHeader](#)

This function returns a pointer to a null terminated string which contains the value of the specified header field in the current message part. For languages like Visual Basic, it is recommended that you use the **MimeGetMessageHeaderEx** function instead.

[MimeGetMessageHeaderEx](#)

This function copies the value of a header field into a null terminated string buffer that you provide. This function also allows you to specify the message part in a multipart message. To return the value of the common header fields such as "From", "To" and "Subject", you should specify a message part of zero.

[MimeGetFirstMessageHeader](#)

This function returns the value of the first header defined in the current message part, copying it into the string buffer that you provide. This is used in conjunction with the [MimeGetNextMessageHeader](#) function to enumerate all of the headers that have been defined.

[MimeGetNextMessageHeader](#)

This function returns the value of the next header defined in the current message part. It should be called in a loop until it returns a value of zero (False) which indicates that the last message header has been returned.

[MimeSetMessageHeader](#)

Set a message header field to the specified value in the current message part. If the value is a null pointer or empty string, the message header will be deleted from the message.

[MimeSetMessageHeaderEx](#)

Set a message header field to the specified value. This version of the function allows you to also specify the message part, rather than just using the current message part. To return the value of the common header fields such as "From", "To" and "Subject", you should specify a message part of zero. If the value is a null pointer or empty string, the message header will be deleted from the message.

[MimeDeleteMessageHeader](#)

Delete the specified message header from the current message part.

Message Contents

The content or body of a message contains the text that is to be read or processed by the recipient. It may be a simple, plain text message or it may be more complex, such as a combination of plain and styled HTML text or the data for a file attachment. The library provides complete access to the contents of the message, enabling the application to modify, extract, replace or delete specific sections of the message.

[MimeGetMessageText](#)

Copy the text from the body of the current message part into a null terminated string buffer. The offset and length of the text being copied can be specified, which enables you to return just a portion of the message. For example, it would be possible to return just the first 1K byte of a text message.

[MimeSetMessageText](#)

Set or replaces the text in the body of the current message part. You can specify an offset in this function, enabling you to replace just portions of the message.

[MimeAppendMessageText](#)

Append text to the body of the current message part. Instead of replacing existing text, the new text will be added to the end of the message.

Multipart Messages

Most typical messages contain a single part, which consists of the message headers followed by the contents of the message. However, when files are attached to a message or alternative content types such as HTML are used, a more complex multipart message is required. With a multipart message, the contents of the message are split into logical sections with each section containing a specific part of the message. For example, when a file is attached to a message, one part of the message contains the text to be read by the recipient and another part contains the data for the file.

The first of a multipart message is called part 0, and contains the main header block. This is what defines the headers that you are most familiar with, such as "From", "To" and "Subject". The body of this message part is typically a plain text message that indicates that this is a multipart message. This is done for the benefit of older mail clients that cannot parse MIME messages correctly. Next part, part 1, typically contains the actual body of the message that would be displayed by the mail client. Additional parts may contain file attachments and other information. In the case of a multipart message that contains both plain and styled HTML text versions of a message, part 1 is typically the plain text version of the message while part 2 contains the HTML version. The mail client can then make a decision based on its own configuration as to which version of the message it displays.

[MimeGetMessagePartCount](#)

Return the number of parts in the message. A simple message will contain just one part. This can be used to determine if the message is a MIME multipart message, with a value greater than one indicating that it is.

[MimeGetMessagePart](#)

Return the current message part. Each message has the concept of a "current message part", which is the default message part that the other functions will use. When a message is first created, the current message part is zero. The simplest way to think of message parts is as a zero-based index into an array into sections of the message.

[MimeSetMessagePart](#)

Change the current message part. Once the current message part has been set, it will remain as the current part until explicitly changed or until a new message part is created.

[MimeCreateMessagePart](#)

Create a new, empty message part. If the message was not originally a multipart message, it will be restructured into one. Otherwise, the new part is simply added to the end of the message. This function will cause the current message part to change to the new part that was just created.

[MimeCreateMessagePartEx](#)

Create a new message part using the specified character set and text. If the message was not originally a multipart message, it will be restructured into one. Otherwise, the new part is simply added to the end of the message. This function will cause the current message part to change to the new part that was just created.

[MimeDeleteMessagePart](#)

Delete the message part from the message. If the message part is in the middle of the message, it will cause the subsequent parts of the message to be reordered. You should not delete part zero to delete a message; use the **MimeDeleteMessage** function instead.

Importing Messages

The library can be used to import existing messages, either from memory or from a file. Once the message has been parsed, the application can examine or modify specific parts of the message. The following functions are provided to import the contents of a message:

[MimeImportMessage](#)

The simplest method of importing a message, this function reads the contents of the specified file and imports it into the current message. This function is typically called immediately after `MimeCreateMessage` to load a file into a new message context.

[MimeImportMessageEx](#)

This version of the function enables you to import the message from a file, the system clipboard, a pointer to a memory buffer or a global memory handle. This is typically used to import messages that were retrieved using the POP3 or IMAP protocols by passing a pointer to the buffer that contains the data returned by the server.

Exporting Messages

After a message has been created or modified, it can be exported to a file or to memory. Exporting the message to a memory buffer is particularly useful when using the library with another one of the SocketTools libraries. For example, the contents of a message can be exported to memory, and that memory address can be passed to the Simple Mail Transfer Protocol (SMTP) library for delivery to the recipient. The following functions are provided to export the contents of a message:

[MimeExportMessage](#)

This function exports the current message to a file. When using this function, only certain headers are exported and they may be reordered. To force all headers to be included in the message or to preserve the order of the headers, use the extended version of this function.

[MimeExportMessageEx](#)

This extended version of the function exports the message to a file, the system clipboard, a memory buffer or returns a handle to a global memory buffer that contains the message. There are additional options that enable you to control what headers are exported, and whether the order the headers were set should be preserved.

File Attachments

In addition to simple text messages, one or more files can be attached to a message. The process of attaching a file involves creating a multipart message, encoding the contents of the file and then including that encoded data in the message. The following functions are provided to manage files attached to the message, as well as attach files to an existing message:

[MimeAttachFile](#)

This function attaches the contents of the file to the message. The file will be attached using the specified encoding algorithm and will become the current message part. If the message is not a multipart message, it will be converted to one; if it already is a multipart message, the attachment will be added to the end of the message.

[MimeAttachData](#)

This function works in similar fashion to **MimeAttachFile**, except that instead of the contents of a file, the data in a memory buffer will be attached to the message. If the message is not a multipart message, it will be converted to one; if it already is a multipart message, the attachment will be added to the end of the message.

[MimeAttachImage](#)

This function attaches an inline image file to the message. It is similar to the **MimeAttachFile** function, except that the image is designed to be referenced as an embedded graphic in an HTML message. This function will automatically set the correct header values for an inline image attachment, and enables the developer to specify a content ID which is used in the HTML message.

[MimeGetAttachedFileName](#)

Return the name of a file attachment in the current message part. This function serves two

purposes, to determine if the current message part contains a file attachment, and if so, what file name should be used when extracting that attachment.

[MimeExtractFile](#)

Extract the file attachment in the current message part, storing the contents in a file. The attachment will automatically be decoded if necessary. This function also recognizes uuencoded attachments that are embedded directly in the body of the message, rather than using the standard MIME format.

Mail Addresses

The library has functions which are designed to make it easier to work with email addresses. Addresses are typically in the format of "user@domain.com" however additional information can be included with the address, such as the user's name or other comments that aren't part of the address itself. The library can parse these addresses for you, returning them in a format that is suitable for use with other protocols such as the SMTP library.

[MimeParseAddress](#)

Parse an email address that may include an address without a domain name or comments in the address, such as the user's name. For example, the From header field may return an address like "Joe Smith <joe@example.com>"; this function would parse the address and return "joe@example.com", the actual address for the user.

[MimeEnumMessageRecipients](#)

It is common for certain headers to contain multiple addresses separated by a comma. These addresses may also include comments such as the user's name. This function parses a string and returns a list of valid addresses. For example, the To header field may contain "Tom Jones <tom@example.com>, Jerry Lewis <jerry@example.com>"; this function would return "tom@example.com" and "jerry@example.com" as the two addresses listed.

Message Storage

The library has a collection of functions which makes it simple for an application to store a group of messages together in a single file, search for and retrieve specific message. The collection of messages is referred to as a "message store" and messages may either be stored in a plaintext format or in a compressed binary format.

[MimeOpenMessageStore](#)

This function is used to open an existing message store or create a new one. The function returns a handle which is used to access the messages in the storage file, and must be closed using the **MimeCloseMessageStore** function. The storage files may either be plaintext, or stored in a compressed format. It also supports opening storage files in the UNIX mbox format.

[MimeGetStoredMessageCount](#)

This function returns the number of messages that currently in the message store. Each message is referred to by an integer which is its index into the storage file. The first stored message has a value of 1, and increments with each additional message in the storage file.

[MimeFindStoredMessage](#)

An application can search the message store for messages that match any header value. Searches can be complete or partial, and may be case-sensitive or case-insensitive. For example, this function can be used to enumerate all of the messages in the storage file that were sent by a specific user or match a specific subject.

[MimeGetStoredMessage](#)

This function returns a handle to the message in the storage file, and can be used with any of the other functions in the Mail Message API. The application can also request its own private copy of

the message, which it can modify independently of what is currently in the message store.

[MimeStoreMessage](#)

This function stores a new message, specified by its message handle, to the open message store. Note that the message store must be opened for write access, and this function will always append the message to the storage file. The new message index is returned to the caller.

[MimeDeleteStoredMessage](#)

This function flags a message for deletion from the message store. Once a message has been flagged for deletion, it may no longer be accessed by the application. When the storage file is closed, the contents of the deleted message will be removed from the file.

[MimeReplaceStoredMessage](#)

This function replaces an existing message in the storage file, overwriting it with the specified message. Unlike many of the other functions which do not permit the application to reference a deleted message, this function can be used to replace a previously deleted message.

[MimeCloseMessageStore](#)

The message store must be closed when the application has finished accessing it. The storage file is updated with any changes, all deleted messages are purged and the handle to the open file is closed. If the storage file is locked for exclusive access, this function will release that lock, allowing another process to open the file.

Network News Transfer Protocol

The Network News Transfer Protocol (NNTP) library enables applications to access a news server, list the available newsgroups, retrieve articles and post new articles. It is common for this library to be used in conjunction with the Mail Message library to construct the articles, since a news article uses the same general format as an email message.

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

[NntpInitialize](#)

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

[NntpConnect](#)

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

[NntpAuthenticate](#)

Provide a user name and password to authenticate the client session. This should only be used if required by the server. Not all news servers require authentication, and some only require authentication when posting articles. If you attempt to perform a function that requires authentication, an error will be returned that indicates you should authenticate and then retry the operation.

[NntpDisconnect](#)

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

[NntpUninitialize](#)

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

Newsgroups

News articles are posted in hierarchical groups, similar to how files are stored in folders. Each level in the newsgroup hierarchy is separated by a period, so newsgroup names look like microsoft.public.vc. This is Microsoft's newsgroup for articles about Visual C++ programming. Additional subgroups are used to further narrow the topic; for example, there's the microsoft.public.vc.3rdparty newsgroup for third party tools and components for Visual C++, and the microsoft.public.vc.atl newsgroup which discusses issues related to the Active Template Library. The NNTP API provides the following functions for accessing newsgroups on the server:

[NntpListGroups](#)

This function requests that the server return a list of all of the newsgroups that are available. If the function is successful, the application should call the `NntpGetFirstGroup` function to begin processing the group list.

[NntpListNewGroups](#)

This function is similar to the **`NntpListGroups`** function in that it requests the server to return a list of available newsgroups. However, the application can request that only groups which were created since a specific date should be returned. This allows the application to maintain a list of newsgroups on the local system, and then use this function to periodically update that list based on the date it was last modified.

[NntpGetFirstGroup](#)

This function is used in conjunction with the **NntpListGroup** or **NntpListNewGroups** functions to enumerate the newsgroups that are available on the server. Information about the newsgroup is returned in a structure, including the name of the group, the first available article number and the last available article in the group.

[NntpGetNextGroup](#)

This function is used to return information about the next newsgroup in the list. It should be called in a loop until it returns zero (False).

[NntpSelectGroup](#)

This function is used to select a newsgroup as the current group. Once selected, the application has access to the articles in that newsgroup.

[NntpGetGroupName](#)

Return the name of the currently selected newsgroup.

[NntpGetGroupTitle](#)

Return a description of the currently selected newsgroup. Note that not all newsgroups have associated descriptions, and some servers may not support the extended command which is used to retrieve the description.

News Articles

News articles are the messages posted to one or more newsgroups. Articles are referenced by their article number, which is a value assigned by the news server. These articles have a structure that is the same as an email message, with some slightly different headers. Because of this, you can use the Mail Message API to parse articles that you retrieve, as well as create new articles to post to the server. The following functions are used to access and create news articles:

[NntpListArticles](#)

This function requests that the server return a list of articles that are available in the current newsgroup. The application can request that all articles be returned, or only those articles which fall into a certain range of article numbers.

[NntpGetFirstArticle](#)

This is the first function that should be called after the **NntpListArticles** function. It will return information about the first article in the list. Article information is returned in a structure which includes information such as the article ID, size, subject, author and date that the article was posted.

[NntpGetNextArticle](#)

This function returns the information about the next article in the list. It should be called in a loop until the function returns zero (False).

[NntpGetArticleRange](#)

Return the range of articles that are available in the currently selected newsgroup. These are the first and last valid article numbers that can be used to retrieve an article from the server. It is important to keep in mind that there is no requirement that articles be stored contiguously with no gaps in between them. For example, say the first available article number in the newsgroup is 101 and the last available article number is 120; it does not necessarily mean that there are 20 available articles. Articles 112 and 118 may have been removed, in which case your application would get an error when trying to access them. The inability to access an article within the article range should not be considered a fatal error; the program should simply move on to the next message.

[NntpGetArticle](#)

Retrieve an article from the server, storing the contents in memory. This can be used to process

the contents of an article without the overhead of storing it in a file on the local system.

[NntpStoreArticle](#)

Retrieve an article from the server and store it in a file on the local system.

[NntpPostArticle](#)

This function posts an article to one or more newsgroups on the server. A newsgroup article is similar to an email message, and the MIME API may be used to create the article headers and body. One important difference is that the message must contain a header named "Newsgroups" with the value set to the newsgroup or newsgroups that the article should be posted to; multiple newsgroups should be separated by commas. If this header is not defined, the posting will be rejected by the server and the function will return an error. You should also be aware that some servers limit the number of newsgroups that a message can be posted to. When an article is posted to more than one newsgroup at a time, this is called cross-posting. Current convention says that an article should not be cross-posted to more than five newsgroups at a time. Also keep in mind that multi-posting (posting the same article to different newsgroups separately) is generally discouraged and should never be done on USENET.

Attaching Files

It is possible to attach files to newsgroup articles; however it should only be done if it is considered appropriate for the group. Many newsgroups have their own acceptable use policies which determine whether or not file attachments, particularly large binary files, are acceptable. If the newsgroup accepts attachments, you can use one of several methods for posting files. It is recommended that you use the [File Encoding](#) API to handle the actual encoding of the data.

Uuencode

A uuencoded file attachment is included directly in the body of the message. Because the MIME API creates a multipart message even when uuencoding is specified, the File Encoding API should be used to encode the data and then it should be included in the main body of the message.

Base64

A Base64 file attachment has the same structure as what is used by email messages. This requires that a multipart message be created, with the encoded data attached as a part of the message. You can use the MIME API to create this kind of message. Note that not all third-party newsreaders correctly handle multipart messages.

yEnc

A popular encoding method used on USENET is called yEnc. Similar to uuencoded attachments, the file data is part of the body of the message. The File Encoding API should be used to encode the data and then it should be included in the main body of the message.

News Feed Library

Really Simple Syndication (RSS) is a collection of standardized formats that are used to publish information about content that is frequently changed. A news feed is published in XML format, which contains one or more items that includes summary text, hyperlinks to source content and additional metadata that is used to describe the item. News feeds can be used for a variety of purposes, including providing updates for weblogs, news headlines, video and audio content. RSS can also be used for other purposes, such as a software updates, where new updates are listed as items in the feed.

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

The first step your application must take is to initialize the library, which will load the required system libraries and initialize the internal data structures that are used. You must call the initialization function before attempting to call any other function in the library.

[RssInitialize](#)

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

[RssUninitialize](#)

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

News Channels

A news feed consists of a channel that contains each of the news feed items. The news channel is represented in the library by an **HCHANNEL** handle. This handle is used with the other functions to reference the specific news feed channel. Information about the news feed, such as the title of the channel and the date it was last modified, is returned in an **RSSCHANNEL** structure.

[RssOpenFeed](#)

Open the channel by specifying a URL to the resource that contains the news. The URL can identify a remote feed that is downloaded using the HTTP or HTTPS protocols, or it can be a file on the local system or network.

[RssParseFeed](#)

Parse a string buffer that contains a news feed. This function is similar to the **RssOpenFeed** function, however it used to parse a string that contains the news feed. This function would typically be used when the feed content is obtained from a different source, such as a database or by using a different protocol. For example, the news feed could be downloaded using the FTP API and then passed to this function.

[RssCloseFeed](#)

This function closes the handle that was allocated by a previous call to the **RssOpenFeed** or **RssParseFeed** function. When the information in a news feed is no longer needed, this function must be called to release the resources allocated to process the feed. After this function is called, the handle is no longer valid.

[RssValidateFeed](#)

This function is used to validate the contents of a news feed, ensuring that it is structured correctly.

The function will return information about the feed, including the number of news items in the feed and the date that it was last modified. The news feed can be specified either as a URL to a remote resource or as a file on the local system or network.

[RssStoreFeed](#)

This function is used to store a news feed as a file on the local system. This is typically used to cache the contents of a news feed or to track the changes made to the feed over time. It is recommended that the application periodically check the publication date of the feed to ensure that they have current version.

News Items

News feed items are identified by a numeric value called the item ID. This is used with other functions to return information about a specific news item. The first item in a news feed has an ID of one and it increments for each additional item in the feed, although it is recommended that applications treat this an opaque value. Functions are provided which will return the number of items in a news feed, information about each item and enable you to easily enumerate each of the items in the feed. Information about a particular news item is returned in an [RSSCHANNELITEM](#) structure.

[RssGetItemCount](#)

This function will return the number of news items in the feed.

[RssGetFirstItem](#) / [RssGetNextItem](#)

These functions are used to enumerate the items in the news feed. Details about each news item is returned in an **RSSCHANNELITEM** structure which contains information such as the title, description and author of the news item.

[RssGetItem](#)

This function is used to return information about a specific news item based on the item ID. When the function returns, it will populate an **RSSCHANNELITEM** structure. Although this function can be used to effectively enumerate all of the news feed items by starting with an item ID value of one, it's recommended that you use the **RssGetFirstItem** and **RssGetNextItem** functions instead. Your application should never make an assumption about the actual value of the item ID because there's no guarantee that future versions of the library will assign item IDs sequentially. Best practices dictate that the **RssGetItem** function should only be called using an item ID that was previously obtained by a call to either the **RssGetFirstItem**, **RssGetNextItem** or **RssFindItem** functions.

[RssFindItem](#)

This function is used to search the feed channel for a specific item, based either on its GUID, title, link or publication date. When searching for a specific item, only searches by GUID are guaranteed to return a unique news item. However, since not all news feeds may provide GUIDs for their news items, additional search criteria can be used when necessary. If the function is successful, it will populate an **RSSCHANNELITEM** structure with information about the news feed item.

[RssGetItemText](#)

This function is used to return a copy of the news item description based on the item ID. Internally, this function calls **RssGetItem** and then copies the item description to the string buffer that is provided by the caller. Note that it is the responsibility of the application to display the text in the appropriate format. Most news feeds will either use plain text or HTML formatted text for the item description.

Post Office Protocol

The Post Office Protocol (POP3) library enables an application to retrieve a user's mail messages and store them on the local system. The POP3 API provides support for all of the standard functionality such as listing and downloading messages, as well as extended features such as the ability to retrieve only the headers for a message or just specific header values. The library also has functions for changing the user's password and sending messages if they are supported by the server.

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

PopInitialize

Initialize the library and load the Windows Sockets library for the current process. This must be the first function call the application makes before calling the other POP3 API functions.

PopConnect

Establish a connection to the POP3 server. This function will return a handle to a client session which is used in subsequent calls to the POP3 API.

PopLogin

Authenticate yourself to the server using a username and password. This function should be called immediately after the connection has been established to the server. You can specify either the standard authentication method, or the APOP authentication method if required by the server.

PopDisconnect

Disconnect from the POP3 server and release any resources that have been allocated for the client session. After this function is called, the client handle is no longer valid.

PopUninitialize

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

Managing Messages

There are functions in the POP3 library for managing messages which enables the application to list, delete and retrieve messages stored on the server. Messages are identified by a number, starting with one for the first message in the mailbox. The most typical operation for a POP3 client is to retrieve each message, store it on the local system and then delete the message from the server. Any processing that is done on the message would then be done on the local copy.

PopGetMessageCount

Return the number of messages available for retrieval. There are two values the application should use. One is the number of currently available messages and the other is the last valid message number. As messages are deleted from the server, the total number of available messages will decrease; however, the last available message number will remain constant.

PopGetMessageCountEx

An extended version of the **PopGetMessageCount** function, this function will return the number of available messages, along with the last available message number and the total size of all messages in the mailbox.

PopGetMessage

Retrieve a message from the server, storing the contents in memory. This can be used to process the contents of a message without the overhead of storing it in a file on the local system.

PopStoreMessage

This function downloads a complete message and stores it as a text file on the local system.

[PopDeleteMessage](#)

Mark the message for deletion. When the connection with the server is closed, the message will be removed from the user's inbox. An important difference between the POP3 and IMAP protocols is that when a message is marked as deleted on a POP3 server, that message can no longer be accessed. An attempt to retrieve a message after it has been marked for deletion will result in an error. The only way to undelete a message once it has been deleted is to terminate the connection with the server by calling the **PopReset** function instead of calling **PopDisconnect**.

[PopGetMessageSize](#)

This function returns the size of the message in bytes. One thing to be aware of when using this function is that some servers will only return approximate message sizes. In addition, because of the difference between the end-of-line characters on UNIX and Windows systems, the size reported by the server may not be the actual size of the message when stored on the local system. Therefore, the application should not depend on this value as an absolute. For example, it should not use this value to determine the maximum number of bytes to read from the server; instead, it should read until the server indicates that the end of the message has been reached.

Message Headers

The POP3 API also includes functions which enable the application to access just the headers for a message. This can be useful if the program doesn't want to incur the overhead of downloading the entire message contents. The following functions can be used to examine the headers in a message:

[PopGetMessageHeaders](#)

This function returns the complete set of headers for the specified message. If your program has to process multiple header fields, this is the most efficient method to use. It is possible to retrieve specific header values, however not all servers support that option and it is somewhat slower because it involves sending individual commands to request each value.

[PopGetHeaderValue](#)

This function returns the value for a specific header field in a message. This function does not require that you parse the message headers; however it does incur additional overhead. It is also important to note that not all servers support the command that is used to request the header value. If this function fails with the error that the feature is not supported, you should use the **PopGetMessageHeaders** function instead.

[PopGetMessageId](#)

This function returns the value of the Message-ID header in the specified message. This is a unique string that is used to identify the message. Note that it is not the same as the UID value returned by the POP3 server.

[PopGetMessageUid](#)

This function returns the unique ID (UID) that the server has associated with the message. The UID can be used by an application to track whether or not it has previously viewed the message. Unlike the message number, which can change between client sessions, the message UID is guaranteed to be the same value across sessions until the message is deleted.

[PopGetMessageSender](#)

This function returns the email address of the person who sent the message. This function requires that the server support extended POP3 commands. If the server does not support the command used to retrieve the sender, it will return a value of zero. Applications should not depend on this function returning a valid address. Typically it is used for informational purposes, such as displaying the sender to the user as a message is being retrieved.

Remote Command Protocol

The Remote Command protocol enables an application to execute commands on a server, with the output of the command returned to the client. The SocketTools library actually implements three related protocols: rexec, rshell and rlogin. The choice of protocols is determined by the port that is selected when a connection is established.

Rexec

The rexec protocol enables a client application to execute a command on a server. Output from the command is returned to the client and the connection is closed when the command terminates. The client connects on port 512 and must provide a user name and password to authenticate the session.

Rshell

The rshell protocol is similar to rexec in that it enables a client to execute a command on a server. Output from the command is returned to the client and the connection is closed when the command terminates. The client connects on port 514 and must provide a user name. The primary difference between the rexec and rshell protocols is that rshell does not require a password. Instead, it uses what is called "host equivalence" to determine if the client is permitted to execute commands as that user. On a UNIX based operating system, host equivalence is controlled by the /etc/hosts.equiv and the .rhosts file in the user's home directory. These files list the host names and user names which are permitted to execute commands using the rshell protocol. Consult your operating system manual pages for more information about how to configure host equivalence.

Rlogin

The rlogin protocol is similar to Telnet in that it provides an interactive terminal session. The connection is closed when the user logs out or the shell process on the server is terminated. The client connects on port 513 and must provide a user name and terminal type. If there is an entry in the host equivalence tables for the user and local host, then the client will be automatically logged in and provided with a shell prompt. If there is no host equivalence, the client will be prompted for a password. The terminal emulation library can be used to provide ANSI or DEC VT-220 emulation services if needed.

An important consideration when deciding whether to use rexec, rshell or rlogin is how the server is configured and the type of command being executed. If there is no entry for the local host in the server's host equivalence tables, then the rexec command should be used instead of rshell.

When using rexec or rshell, it is important to keep in mind that although the command is executed with the privileges of the specified user, that user is not actually logged in. The user's login script is not executed and the program will not inherit the user's normal environment as it would during an interactive session. If you are connecting to a UNIX system, you should not attempt to execute programs which try to put standard input into raw mode; an example of this would be the vi editor. If you are connecting to a Windows system, you should not execute a program which uses a graphical interface. Only programs which read standard input and write to standard output are suitable for use with rexec or rshell.

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

[RshInitialize](#)

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

[RshExecute](#)

Execute the specified command on the server. The rshell or rexec protocol is selected based on the port number that is specified. Output from the command will be returned to the client to be read. When the command terminates, the connection to the server will be closed.

[RshLogin](#)

Establish an interactive login session which is similar to how the Telnet protocol works. If there is no host equivalence with the local host, you will be prompted for a password. Output from the session will be returned to the client, and when the client logs out the connection will be closed.

[RshRead](#)

Read the output generated by the command. Your application would typically call this function in a loop until all of the data has been read or an error occurs.

[RshSearch](#)

Search for a specific sequence of characters in the output returned by the server. The function returns when the sequence is encountered or when a timeout occurs. The data captured up to the point where the character sequence was matched is returned to the caller for processing.

[RshDisconnect](#)

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

[RshUninitialize](#)

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

Secure Shell Protocol

The Secure Shell (SSH) protocol enables an application to establish a secure, interactive terminal session with a server, or execute commands remotely on the server, with the output of the command returned to the client. The SocketTools library supports both version 1.0 and 2.0 of the protocol.

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

[SshInitialize](#)

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

[SshConnect](#)

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

[SshDisconnect](#)

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

[SshUninitialize](#)

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

Connection Options

The SSH protocol has a number of advanced options which can be specified when establishing a connection. To provide additional information to the connection function, the [SSH_OPTIONDATA](#) structure is used. The three most commonly used options specify if the connection will be used for an interactive terminal session or to execute a command on the server, and if a proxy server should be used when connecting to the server:

SSH_OPTION_TERMINAL

This option specifies the client session will use terminal emulation and the [SSH_OPTIONDATA](#) structure specifies the characteristics of the virtual terminal. This enables the caller to specify the dimensions of the virtual display (in columns and rows) and the type of terminal that will be emulated. If this option is omitted, the session will default to a virtual display that is 80 columns, 25 rows.

SSH_OPTION_COMMAND

This option specifies the client session will be used to issue a command that is executed on the server, and the output will be returned to the caller. If this option is specified, the session will not be interactive and no pseudo-terminal is created for the client. The *szCommandLine* member of the [SSH_OPTIONDATA](#) structure specifies the command string that will be sent to the server.

SSH_OPTION_PROXYSERVER

This option specifies the client should establish a connection through a proxy server. The two protocols that are supported are [SSH_PROXY_HTTP](#) and [SSH_PROXY_TELNET](#), which specifies the protocol that the proxy connection is created through. The proxy-related members of the [SSH_OPTIONDATA](#) structure should be set to the appropriate values.

Input and Output

Once your application has connected to the server, any output generated by a program on the server will be sent as data for you to read. Any input to the program is sent by your application and received and processed by the server. The following functions are used:

SshPeek

Reads any data that has been sent by the server and copies it to the buffer provided by the caller, but it does not remove the data from the internal receive buffer. This function can be used to examine the contents of the receive buffer and make decisions about how to process the data.

SshRead

Reads any data that has been sent by the server and copies it to a byte array buffer provided by your application. If the server closes the connection, the function will return a value of zero and the client can disconnect from the server at that point.

SshReadLine

Reads a line of text from the server, and returns it as a null terminated string. This function is useful for reading output from the server one line at a time. The function recognizes both UNIX and Windows end-of-line conventions, and will cause the application to block until a complete line of text has been read.

SshWrite

Send data to the server which will be received as input to the program. Your application provides the function with a byte array buffer that contains the data, and an integer value that specifies the number of bytes in the buffer.

SshWriteLine

Send data to the server as a line of text, terminated with a carriage-return and newline character. Note that your application should not specify the end-of-line characters, they are automatically sent to the server. This function will cause the application to block until the complete line of text has been written.

Command Processing

The SSH protocol can be used to connect to a server, log in and execute one or more commands, process the output from those commands and display it to an end-user using a graphical interface. The user never needs to see or interact with the actual terminal session. The SSH API provides functions which can simplify this kind of application, reducing the amount of code needed to process the data stream returned by the server.

SshExecute

This function executes a command on a server and copies the output to a user-specified buffer, with the exit code for the remote program as the function's return value. This is a convenience function that enables you to execute a remote command in a single call, without having to write the code to establish the connection and read the output.

SshGetExitCode

This function returns the exit code for the program that was executing on the server. It should be called after all of the data has been read and the server has closed the connection, which is indicated by the **SshRead** function returning a value of zero.

SshSearch

This function is used to search for a specific character or sequence of characters in the data stream returned by the server. The library will accumulate all of the data received up to the point where the character sequence is encountered. This can be used to capture all of the output from a command, or search for specific results returned by the command as it executes on the server.

Simple Mail Transfer Protocol

The Simple Mail Transfer Protocol (SMTP) enables applications to deliver email messages to one or more recipients. The library provides an API for addressing and delivering messages, and extended features such as user authentication and delivery status notification. This library is typically used in conjunction with the Mail Message library to create the messages, and the Domain Name Service library to determine what servers are responsible for accepting mail for a specific user.

Mail Exchanges

When a message is delivered to a user, the application must determine what mail server is responsible for accepting messages for that user. This can be accomplished using the Domain Name Services (DNS) protocol, a protocol that is most commonly used to resolve host names such as `www.microsoft.com` into Internet addresses. This is typically accomplished by sending a request to a nameserver, a computer system that provides domain name services. In addition to resolving host names, nameservers can also provide information about those servers which are responsible for accepting mail for a given domain. There can be multiple servers which process mail for a domain with each server assigned a priority as part of their mail exchange (MX) record. If there is no mail exchange record for a domain, then the domain name itself is used.

To deliver a message directly to the recipient, you must examine the recipient address and request the list of mail exchanges for that user's domain. Using the DNS API, this is done by calling the **DnsEnumMailExchanges** function. If the recipient address is `joe@example.com`, you would want to enumerate the mail exchanges for the `example.com` domain. This will give you the name of the servers that will accept mail for users in that domain. For example, the function may return the host name `mail.example.com` as the name of the server which will accept mail for users in the `example.com` domain. Note that it is possible that one or more of the mail exchanges for a domain may not be in the recipient domain itself. In other words, it is possible that `smtp.othercorp.net` could be returned as a mail exchange for `example.com`. This is frequently the case when another organization is forwarding mail for that domain.

Therefore, there are three general steps that you must take when delivering mail directly to the recipient:

1. Parse the address of each recipient in the message. If you are using the MIME API, the **MimeEnumMessageRecipients** function can be helpful in extracting all of the recipient addresses. Everything after the atsign (@) in the address is the domain portion of that address.
2. Perform an MX record lookup using the DNS API function **DnsEnumMailExchanges** and specifying the recipient's domain. The function will return the name of the servers responsible for accepting mail for that user. If there are more than one server, they will be returned in order of their relative priority, with the highest priority server being returned first. This means that you should attempt to connect to those servers in the order that they are returned by the function.
3. Attempt to connect to the first server returned by the **DnsEnumMailExchanges** function. The connection should be on the default port, and you should not attempt to use any authentication. If the server accepts the connection, then use the **SmtpSendMessage** function to deliver the message. If the connection is rejected or the message is not accepted, attempt to connect to the next mail exchange server until all servers have been tried.
4. If no mail exchange servers were returned by **DnsEnumMailExchanges**, or you could not connect to any of them, attempt to connect to the domain specified in the address using the default port. If the connection succeeds, then deliver the message. If you cannot connect or the message is not accepted, then report to the user that the message could not be delivered.

One last important consideration is that many Internet Service Providers now block outbound connections

on port 25 to any mail servers other than their own. If you are unable to establish any connections, either with the error that the connection was refused or it consistently times out, contact your ISP to determine if port 25 is being blocked as an anti-spam measure. If this is the case, it will be required that you relay all messages through their mail servers or use an alternate port number.

Relay Servers

In some situations it may not be possible to send mail directly to the server that accepts mail for a given domain. The two most common situations are corporate networks which have centralized servers that are responsible for delivering and forwarding messages, or an Internet Service Provider (ISP) which specifically blocks access to all mail servers other than their own. This is usually done as either a security measure or as a means to inhibit users from sending unsolicited commercial email messages. If the standard SMTP port is being blocked, then any connection attempts will either fail immediately with an error that the server is unreachable, or the connections will simply time-out. In either case, a relay server must be specified in order to send email messages.

A relay server is a system which will accept messages addressed to users who may be in a different domain, and will relay those messages to the appropriate server that does accept mail for the domain. Using a relay server is generally easier than sending messages directly to the recipient. In order to send a message through a relay, you need to perform the following steps:

1. Connect to the relay server as you would normally.
2. Authenticate the client to the server. This may or may not be required, depending on how the server is configured. Some servers may be configured to only require authentication if you are connecting from an IP address that is not recognized as part of that system's network, for example, if you are connecting using a different Internet Service Provider. Others may always require authentication. Check with the server administrator if necessary to determine if and when authentication is required.
3. Use the **SmtpSendMessage** function to deliver the message to the recipients through the relay server. If there are multiple recipients, you can use the MIME API to enumerate the recipient addresses and then pass them to the **SmtpSendMessage** function.

It is important to note that using a mail server as a relay without the permission of the organization or individual who owns that server may violate Acceptable Use Policies and/or Terms of Service agreements with your service provider. Systems which relay messages from anyone, regardless of whether the message is coming from a recognized domain, are called open relays. Because open relays are often used to send unsolicited email, many administrators block mail that comes from one. It is recommended that users check with their network administrators or Internet service providers to determine if access to external mail servers is restricted and what is the acceptable use policy for relaying messages through their mail servers.

The first step your application must take is to initialize the library, then establish a connection to the server and authenticate the client if necessary. The following functions are available for use by your application:

[SmtpInitialize](#)

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

[SmtpConnect](#)

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

[SmtpAuthenticate](#)

Authenticate the client session to the server using a username and password. This function should be called immediately after the connection has been established to the server. This is typically required if you are attempting to use the mail server as a relay, asking it to forward the message

on to the server that actually accepts email for the recipient. Many Internet Service Providers (ISPs) require that users authenticate prior to sending mail through their servers. You may need to contact the server administrator to determine if authentication is required.

[SmtpDisconnect](#)

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

[SmtpUninitialize](#)

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

Message Delivery

There are two general methods that can be used to deliver messages through the mail server. In most cases, it can be done with a single function call. However, there are some circumstances where it would be more appropriate to perform the transaction in stages. The SMTP API supports both methods.

[SmtpSendMessage](#)

This is the simplest method for sending an email message through the server. You provide the sender and recipient addresses, along with the message contents and the function will submit the message to the server for delivery.

[SmtpCreateMessage](#)

This function begins a transaction in which a message is dynamically composed, addressed and delivered in stages. You provide the sender address and message size to this function, and after it returns you begin the next stage, which is addressing the message.

[SmtpAddRecipient](#)

This function adds a recipient address to the recipient list for the message. This should be called once for each recipient, as well as for any recipients who are to receive "blind copies" of the message. A blind copy is when the message is sent to a recipient, but that recipient's address is not listed in any of the headers of the message; the other recipients will be unaware that the message was delivered to him. Most servers have a limit of approximately 100 recipients per message. It is possible that this function will return an error for a specific recipient address; the address may be malformed or it may not be acceptable for some other reason. This does not mean that the message will be rejected in its entirety, only that the specified recipient is not acceptable.

[SmtpAppendMessage](#)

This function should be called after all of the recipients have been added. It is used to send the contents of the message to the server. It is also possible to use the lower level **SmtpWrite** function to send data directly to the server, however **SmtpAppendMessage** is generally easier to use and can write data from memory, the system clipboard or from a file on disk.

[SmtpCloseMessage](#)

This function is called after the entire message has been sent to the server. This terminates the transaction and the message is submitted for delivery. Note that it is possible for the server to accept the message up to this point and then reject it at this final step due to some restriction, such as the message being too large.

SocketWrench

The SocketWrench library provides an interface to the Windows Sockets API. It was designed to be simpler to use, and to provide functions which eliminate much of the redundant coding common to Windows Sockets programming. Developers who are working in languages other than C or C++ will find SocketWrench to be particularly useful because it does not use many of the complex structures that the Windows Sockets API uses. SocketWrench also supports creating client and server applications which use the SSL and TLS security protocols without any dependencies on third-party security libraries.

The first step your application must take is to initialize the library. After the library has been initialized, the application can either take on the role of a client and establish a connection to a server, or become a server and listen for incoming connections from other clients.

[InetInitialize](#)

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

[InetConnect](#)

Establish a connection with a server. This function will return a handle to the application which can be used to send and receive data. When an application calls this function, it will be acting as a client. For an asynchronous session, use the **InetAsyncConnect** function.

[InetListen](#)

Begin listening for incoming client connections. This function will return a handle which should be passed to the **InetAccept** function to accept any clients which establish a connection. When an application calls this function, it will be acting as a server. For an asynchronous session, use the **InetAsyncListen** function.

[InetAccept](#)

Accept a connection from a client. This function should only be called if the application has previously called **InetListen**. If there is no client waiting to connect at the time this function is called, it will block until a client connects or the timeout period is reached. For an asynchronous session, use the **InetAsyncAccept** function.

[InetUninitialize](#)

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

Input and Output

When a TCP connection is established, data is sent and received as a stream of bytes. The following functions can be used to send and receive data over the socket:

[InetRead](#)

Read data from the socket and copy it to the memory buffer provided by the caller. If the server closes the connection, this function will return zero after all the data has been read. If the function is successful, it will return the actual number of bytes written.

[InetIsReadable](#)

This function is used to determine if there is data available to be read from the socket.

[InetWrite](#)

Write data to the socket. If the function succeeds, the return value is the number of bytes actually written.

[InetIsWritable](#)

This function is used to determine if data can be written to the socket. In most cases this will return

a non-zero value (True), unless the internal socket buffers are full.

Server Interface

The library provides a collection of functions which can be used to easily create a scalable, event-driven multithreaded server application. The server runs on a separate thread in the background, automatically managing the individual client sessions as servers connect and disconnect from the server. The application is notified of events through a callback mechanism, where it can respond to notifications such as a client establishing a connection with the server, or the client sending data to the server. Functions such as **InetRead** and **InetWrite** are used to exchange data with the clients. Because each client session is managed in its own thread, applications can perform calculations, access database resources and so on without worrying about interfering with other client sessions or the application's main UI thread.

[InetServerStart](#)

This function starts the server, creating the background thread and listening for incoming client connections on the specified port number. A socket handle is returned to the caller which is used to query the status of the server and perform other functions such as suspending and restarting the server.

[InetEnumServerClients](#)

This function enables the server application to determine the number of active client sessions, and obtain socket handles for each client that is connected to the server.

[InetServerBroadcast](#)

Broadcasts data to each of the clients that are connected to the server. This can be useful when the application needs to send the same data to each active client session, such as broadcasting a shutdown message when the server is about to be terminated.

[InetServerThrottle](#)

This function is used to control the maximum number of clients that may connect to the server, the maximum number of clients that can connect from a single IP address and the rate at which the server will accept client connections. By default, there are no limits on the number of active client sessions and connections are accepted immediately. This function can be useful in preventing denial-of-service attacks where the attacker attempts to flood the server with connection attempts.

[InetServerSuspend](#)

This function instructs the server to temporarily suspend accepting new client connections. Existing connections are unaffected, and any incoming client connections are queued until the server is resumed. It is not recommended that you leave a server in a suspended state for an extended period of time. Once the connection backlog queue has filled, any subsequent client connections will be automatically rejected.

[InetServerResume](#)

This function instructs the server to resume accepting client connections after it was suspended. Any pending client connections are accepted after the server has resumed normal operation.

[InetServerLock](#)

This function locks the server so that only the current thread may interact with the server and the client sessions. This will cause all other client threads to go to sleep, waiting for the server to be unlocked. This should only be used when the server application needs to ensure that no other client threads are performing a network operation. For general purpose synchronization, it is recommended that the application create a critical section rather than lock the entire server. If the server is left in a locked state for an extended period of time, it will cause the server to become non-responsive. If the application has started multiple servers, only one server can be locked at any one time.

[InetServerUnlock](#)

This function unlocks a server that has been previously locked. The threads which manage the client sessions will awaken and resume normal execution.

[InetServerRestart](#)

This function will terminate all active client connections, close the listening socket and re-create a new listening socket bound to the same address and port number. The function will return a socket handle for the new listening socket that should replace the original socket that was allocated using the **InetServerStart** function.

[InetServerStop](#)

This function will terminate all active client connections, close the listening socket and terminate the background thread that manages the server. Any incoming client connections will be refused, and all resources allocated for the server will be released.

Address Conversion

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

[InetGetAddress](#)

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

[InetFormatAddress](#)

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

Host Tables

When resolving a host name or IP address, the library will first search the local system's host table, a file that is used to map host names to addresses. On Windows 95/98 and Windows Me, if the file exists it is usually found in C:\Windows\hosts. On Windows NT and later versions, it is found in C:\Windows\system32\drivers\etc\hosts. Note that the file does not have an extension.

[InetGetDefaultHostFile](#)

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

[InetGetHostFile](#)

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

[InetSetHostFile](#)

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

Host Name Resolution

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

[InetGetHostAddress](#)

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

[InetGetHostName](#)

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

Local Host Information

Several functions are provided to return information about the local host, including its fully qualified domain name, local IP address and the physical MAC address of the primary network adapter.

[InetGetLocalName](#)

Return the fully qualified domain name of the local host, if it has been configured. If the system has not been configured with a domain name, then the machine name is returned instead.

[InetGetLocalAddress](#)

Return the IP address of the local host. If a valid socket handle is provided, then the IP address of the network adapter that was used to establish the connection will be returned. This can be particularly useful for multihomed systems that have more than one adapter and the application needs to know which adapter is being used for the connection.

[InetGetAdapterAddress](#)

Return the IP or MAC address assigned to a network adapter on the local system.

[InetEnumNetworkAddresses](#)

Enumerate the network addresses that are configured for the local host. If the system is multihomed, then the IP address for each network adapter will be returned.

Telnet Protocol

The Telnet Protocol library enables an application to connect to a Telnet server, which provides an interactive terminal session similar to how character based consoles and terminals work. The user can login, enter commands and interact with applications programmatically or in conjunction with the terminal emulation library.

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

TelnetInitialize

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

TelnetConnect

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

TelnetDisconnect

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

TelnetUninitialize

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

Input and Output

Once connected to the Telnet server, any output generated by a program on the server will be sent as data for the client to read. Any input to the program is sent by the client and received and processed by the server. The following functions are used:

TelnetRead

Reads any output that has been generated by the program executing on the server. When the client first connects, the server typically executes a login program that requests the users authenticate themselves by entering a user name and password. Once the user has logged in, they are usually given a command line prompt where they can enter commands to be executed on the server. If the server closes the connection, the **TelnetRead** function will indicate that with an error result and the client can disconnect from the server at that point.

TelnetWrite

Send data to the Telnet server which will be received as input to the program. If the local echo option is enabled, then the client is also responsible for writing the input data to the display device, if there is one. If local echo is not enabled, the server will automatically echo back any characters written as data to be read by the client.

Telnet Modes

Telnet supports several modes of operation and the option negotiation phase, which occurs when a connection is established, is handled automatically by the library. There are two key modes which affect how the client session works:

TELNET_MODE_LOCALECHO

If this mode is enabled, it is the responsibility of the client to echo any data that it is sending to the server. For example, if the character "A" is sent to the server, the application must also send the character "A" to whatever interface the user is interacting with, such as a terminal emulation window. The default mode is for this option to be disabled, which means that the server will echo

back any data that is sent to it.

TELNET_MODE_BINARY

If this mode is enabled, the data between the client and server is not buffered and the high bit is not removed from any characters. If the application is executing a program which uses text mode windowing features (i.e.: it draws boxes on the display) then this mode must be enabled to ensure that the client processes the data correctly and it isn't buffered a line at a time. If this mode is disabled, then the data exchanged between the client and server will be buffered a line at a time and any 8bit characters will be stripped. This mode is enabled by default.

Command Processing

The Telnet protocol can be used to connect to a server, log in and execute one or more commands, process the output from those commands and display it to an end-user using a graphical interface. The user never sees or interacts with the actual terminal session. The Telnet API provides functions which can simplify this kind of application, reducing the amount of code needed to process the data stream returned by the server.

[TelnetLogin](#)

This function is used to automatically log a user in, using the specific user name and password. This function is specifically designed for UNIX based servers or Windows servers which emulate the same basic login sequence.

[TelnetSearch](#)

This function is used to search for a specific character or sequence of characters in the data stream returned by the server. The library will accumulate all of the data received up to the point where the character sequence is encountered. This can be used to capture all of the output from a command, or search for specific results returned by the command as it executes on the server.

Terminal Emulation

The Terminal Emulation library provides a virtual terminal interface for emulating an ANSI or DEC VT-220 compatible character-based terminal. It can be used in conjunction with the Telnet API or the Remote Command API to display the output of commands executed on a server. It can also be used independently of any other networking library, such as providing emulation services for a serial connection.

The first step your application must take is to initialize the library, and then create a virtual display which is attached to a window. The following functions are available for use by your application:

[NvtInitialize](#)

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

[NvtCreateDisplay](#)

Creates a new virtual display and returns a handle which can be used to reference that display. When creating the display, you provide a handle to a window the display will draw on, a handle to a font that will be used to display text characters, the type of emulation and the size of the display.

[NvtDestroyDisplay](#)

Destroy the virtual display, releasing the memory allocated for the handle. After this function returns, the display handle is no longer valid.

[NvtUninitialize](#)

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

Display Management

The library provides functions to manage and update the virtual display, including a number of lower level functions that provide direct access to the display buffer. The most commonly used functions are:

[NvtWriteDisplay](#)

This is the most commonly used method of writing to the display. This function will automatically parse the data being written for escape sequences and update the display appropriately.

[NvtUpdateDisplay](#)

This function updates the window attached to the virtual display. This function should be called whenever a WM_PAINT message is received for that window, indicating that it needs to be redrawn. The window paint message can be handled in one of two general ways. The simplest is to just call **NvtUpdateDisplay** and return. The function will automatically acquire a device context for the window, redraw the portions of the window that need to be updated, and then release the device context. If you use this method, you should not call **BeginPaint** or **EndPaint**, since that is being done for you. If you wish to acquire the device context yourself by calling **BeginPaint**, then you must call **NvtSetDisplayDC** to attach the device context to the virtual display. When **NvtUpdateDisplay** is called, it will use that device context rather than acquiring one itself. After the display has been updated and you've called **EndPaint**, you should call **NvtSetDisplayDC** again, passing the function a null handle to detach the display from the device context that you've created.

[NvtRefreshDisplay](#)

Refresh the virtual display, updating the current cursor position and caret. The library will periodically refresh the display automatically based on its own internal state, but the application can call this if it wishes to force the display to refresh at that time.

[NvtSetDisplayEmulation](#)

This function can be used to change the emulation used by the library. Supported emulation types are standard ANSI, DEC VT-100 and VT-200. It is also possible to specify that no emulation should be used, in which case the application is responsible for handling any control or escape sequences in the data being written to the display.

[NvtResetDisplay](#)

This function can be used to change the handle of the window associated with the display, the font being used and the size of the display. Resetting the display causes the contents of the display to be cleared.

[NvtSetDisplayFont](#)

This function sets the font which is used by the library to draw text on the display window. If you wish to specify a font name and point size instead of creating a font handle, use the **NvtSetDisplayFontName** function instead.

[NvtSetDisplayMode](#)

The emulation library has a number of modes which determines how the data is displayed, as well as controlling aspects of the window itself. This function can be used to enable or disable those modes as necessary. For the list of available modes, please consult the Technical Reference.

[NvtSetDisplayColor](#)

This function controls what colors are used when drawing the background of the display window as well as the default color of the text. In ANSI and VT-220 emulation, the color of individual characters can be specified by using escape sequences. Those colors can be modified by changing the display color map.

[NvtSetDisplayColorMap](#)

This function changes the default colors which are used when escape sequences are used to change the foreground or background color of a character cell. In most cases the default color map will be appropriate, but applications can change the RGB values associated with an entry in the color map if needed. For example, the default value for the color gray is at position 8 in the color map index with an RGB value of 192,192,192. If you wanted to use a darker color, you could change the RGB value 128,128,128.

Cursor Control

There are a number of lower level functions which enable an application to have direct control over cursor positioning, clearing the display and so on. In most cases these functions are called automatically by the library as the result of processing the control and escape sequences found in the data being written to the display. However, an application can call these functions directly in order to manage the display itself. One important thing to keep in mind is that the X,Y positions used by these functions refer to the cursor position in the virtual display and correspond to columns and rows, not pixels.

There is also a slight difference in terminology that you should be aware of when reading the technical reference documentation. In Windows, the term "cursor" is typically used to refer to the mouse pointer, while "caret" is used to refer to the blinking marker that is displayed at the current position in the display. In the documentation for the emulator API, the term "cursor" is used in the same way that it is used for character based terminals, as the marker for the current position in the display. Therefore, in terms of the emulation API, you can think of the cursor and the caret as being synonymous.

[NvtGetCursorPos](#)

Return the current position of the cursor in the virtual display.

[NvtSetCursorPos](#)

Change the current position of the cursor in the virtual display. This function will normalize the X,Y values to be bound by the size of the display. If a scrolling region has been set, the coordinates

will be bound by that region. The upper left corner of the display is 0,0.

[NvtSaveCursor](#)

Saves the current position of the cursor. The cursor position can be restored by calling the **NvtRestoreCursor** function.

[NvtClearDisplay](#)

Clear the contents of the display. You can clear from the start of the display to the current cursor position, from the current position to the end of the display or the entire display.

[NvtDeleteChar](#)

This function deletes a character from the current cursor position, shifting the remaining characters on the line to the left.

[NvtDeleteLine](#)

This function deletes the line at the current cursor position, shifting the remaining lines in the display up.

[NvtEraseChar](#)

This function erases a character at the current cursor position without affecting the characters that follow it on the line.

[NvtEraseLine](#)

This function erases the line at the current cursor position without affecting the lines that follow it.

[NvtInsertChar](#)

Insert a character at the current cursor position, shifting the following characters to the right.

[NvtInsertLine](#)

Insert a blank line at the current cursor position, shifting the following lines down.

[NvtGetDisplayAttributes](#)

Return the current attributes which have been enabled. Possible attributes are reverse, bold, dim, underline, hidden and protected. These correspond to the attributes that are commonly used with character based terminals.

[NvtSetDisplayAttributes](#)

Set the default attributes which are used when characters are written to the display. For example, setting the attribute `NVT_ATTRIBUTE_REVERSE` would cause all subsequent characters to be displayed with the foreground and background colors reversed. This would continue until the display attributes were set back to `NVT_ATTRIBUTE_NORMAL`.

Function Key Mapping

Another aspect of terminal emulation is how function keys and other special keys are handled by the application. The emulation library provides functions which will convert Windows virtual key codes into the escape sequences that are generated by character based terminals.

[NvtTranslateMappedKey](#)

This function translates a virtual key code into the escape sequence which should be sent to the server to emulate pressing that key. For example, if the user presses the F1 key on the keyboard, this will generate a `WM_KEYDOWN` event with `VK_F1` as the virtual key code. This function will translate that key code into the three character escape sequence `ESC O P` (the ASCII codes 27, 79, 80). That sequence of characters should be sent to the server, which will recognize it as the F1 function key being pressed. It is important to note that the different emulation types have different key mappings. Therefore, the server must be set to recognize the same type of terminal that you are emulating. If you have the emulation set as VT-220 but the server thinks that you are emulating a VT-100, it will not recognize some of the escape sequences correctly.

NvtGetMappedKey

Returns the escape sequence associated with a specific function key or special key. There are a total of 50 special keys recognized by the emulator. They consist of the F1 through F12 function keys, the function keys when they are in a shift state, the arrow keys and the keypad keys. Refer to the technical reference for the complete list.

NvtSetMappedKey

Change the escape sequence associated with a specific function key or special key. Changing this value will cause **NvtTranslateMappedKey** to return the new escape sequence when that key is pressed. This function can also be used to restore the default mapping for a key.

Windows Messages

Because the emulation library is closely tied to a display window, it is important to understand how Windows messages should be handled. If you are not familiar with how Windows processes messages at the API level, it is recommended that you review the technical reference material available from Microsoft on user interface programming. Another excellent resource is "Programming Windows" by Charles Petzold. If you are a C++ programmer, the equivalent MFC virtual functions will also be listed here.

WM_CREATE

This message is sent when the window is being created, but before it is displayed. This is typically a good point to call the **NvtCreateDisplay** function to create the virtual display and attach it to the window. You can also call **NvtSetDisplayMode** to set the various display modes, such as whether you want scrollbars to be shown, what kind of caret should be used and so on. In MFC, this can be done in the Create method for the window.

WM_DESTROY

This message is sent when a window is being destroyed. The application should call **NvtDestroyDisplay** at this point, since the window will no longer exist after the event handler returns.

WM_PAINT

This message is sent when the window needs to be redrawn. This is where the **NvtUpdateDisplay** function should be called. If you are using MFC, the code should be placed in the OnPaint method.

WM_HSCROLL and WM_VSCROLL

This message is sent when the user interacts with the window's scrollbars. An application should call the **NvtSetDisplayScrollPos** function to update the scroll position in the virtual display. In MFC, this code should be implemented in the **OnHScroll** and **OnVScroll** methods.

WM_KEYDOWN

This message is sent when the user presses a key. To determine if the key is mapped to a special escape sequence that should be sent to the server, call the **NvtTranslateMappedKey** function. In MFC, this code should be implemented in the **OnKeyDown** method.

WM_SETFOCUS and WM_KILLFOCUS

This message is sent when the window acquires or loses focus. You should call **NvtSetDisplayFocus** so that the library knows that the input focus has changed. In MFC, this code should be implemented in the **OnSetFocus** and **OnKillFocus** methods.

WM_SIZE

This message is sent when the window size changes. The application should call the **NvtResizeDisplay** function so that the virtual display is aware that the window size has changed. In MFC, this code should be implemented in the **OnSize** method.

When writing a terminal emulation program, it will also be necessary to handle other Windows messages

such as WM_CHAR to process keys pressed by the user, which should be sent to the server. It is recommended that you refer to the Terminal sample program which demonstrates how these messages can be handled by an application. It also serves as a good example of how the Terminal Emulation API can be used in conjunction with the Telnet API.

Text Message Library

Short Message Service (SMS) is a text messaging service used by mobile communication devices to exchange brief text messages. Most service providers also provide gateway servers that can be used to send messages to a wireless device on their network using standard email protocols. The Text Message API provides functions that can be used to determine the provider associated with a specific telephone number and send a text message to the device using the provider's mail gateway.

The first step your application must take is to initialize the library, which will load the required system libraries and initialize the internal data structures that are used. You must call the initialization function before attempting to call any other function in the library.

[SmsInitialize](#)

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

[SmsUninitialize](#)

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

Text Messages

Sending a text message is done with a single function call, with two parameters. The first parameter is a pointer to a data structure that identifies the service that will be used to send the message. By default, messages are sent via an SMTP gateway, however the API was designed to be extensible so that additional methods could be integrated into future versions of the library. For example, a third-party company may offer a service that allows messages to be sent using HTTP and that can be added as an additional service type. The second parameter is a pointer to a data structure that contains information about the text message itself.

[SmsSendMessage](#)

This function is used to send the text message. The caller must populate two data structures that contain information about the service provider and the contents of the message itself. The [SMSSERVICE](#) structure is used to provide information about the service being used to send the message, and the [SMSMESSAGE](#) structure is used to provide the function with information about the message.

Service Providers

When a service provider is mentioned in the documentation, typically it is referring to the wireless service provider (also commonly called a "carrier") that is responsible for providing network access for the mobile device. These are identified by name, such as "Verizon Wireless" and "AT&T Mobility". The library has a built-in table of known providers in North America, and can return this information to your application. Note that in some cases, a service provider may also refer to a specific service used to send a text message.

[SmsGetProvider](#)

This function returns information about the wireless service provider that is associated with a specific phone number. This can be used to determine if a phone number is valid, and the default provider that is responsible for that number. Information about the service provider is returned in an [SMSPROVIDER](#) structure.

[SmsEnumProviders](#)

This function enumerates all of the supported wireless service providers and populates an array of structures that contain information about each provider. Typically this is used to update the user interface with a list of known service providers, allowing the end-user to select a specific service provider.

[SmsGetFirstProvider](#) and [SmsGetNextProvider](#)

These functions are also used to enumerate the supported wireless service providers, but they return information about a single provider with each function call. They are primarily designed to be used with languages that don't provide an easy way to work with an array of data structures. The information that is returned is identical to the **SmsEnumProviders** function.

Gateway Servers

A gateway server refers to the server that is responsible for accepting the text message and sending it to the recipient. Currently, this is exclusively used in the context of SMTP gateways where the message is sent to a mail server operated by the wireless service provider.

[SmsGetGateway](#)

This function returns information about the gateway that is responsible for accepting a text message for a specific phone number and forwarding that message to the mobile device. Information about the gateway is returned in an [SMSGATEWAY](#) structure.

Time Protocol

The Time protocol library enables an application to retrieve the current time from a server, and optionally synchronize the local system time using that value. The first step that your application must take is to initialize the library. After the library has been initialized, the application can request the current time from a system and update the local system clock if necessary.

TimeInitialize

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

GetNetworkTime

Return the current time from a server. The time is expressed as a 32-bit integer value which represents the number of seconds since midnight, 1 January 1900 UTC.

UpdateLocalTime

Update the local system time with the value returned by **GetNetworkTime**. This function requires that the current user have the appropriate permissions to modify the system time or the function will fail.

TimeUninitialize

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

Time Conversion

Windows applications typically use a structure called **SYSTEMTIME** to represent date and time values. The library has two functions which will enable you to convert between the value returned by **GetNetworkTime** and the **SYSTEMTIME** structure.

ConvertNetworkTime

This function will convert the value returned by **GetNetworkTime** into a **SYSTEMTIME** structure, adjusting for the local timezone if required.

ConvertSystemTime

This function will convert a **SYSTEMTIME** structure into a 32-bit integer value. This value may be passed to the **UpdateLocalTime** function to update the local system clock.

Whois Protocol

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

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

[WhoisInitialize](#)

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

[WhoisConnect](#)

Connect to the server, using either a host name or IP address. This function returns a client handle which is used in subsequent calls to the library.

[WhoisSearch](#)

Perform a search for a specific domain. The server will return the results of the search as a text document which provides registration information for that domain. It is important to note that different registrars may use different formats when returning the data, and not all servers return the same type of information.

[WhoisRead](#)

Read the data returned by the server. Your application would typically call this function in a loop until all of the data has been read or an error occurs.

[WhoisDisconnect](#)

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

[WhoisUninitialize](#)

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

SocketTools Class Library Reference

- Domain Name Service Class
- File Transfer Protocol Class
- File Transfer Server Class
- Hypertext Transfer Protocol Class
- Hypertext Transfer Server Class
- Internet Control Message Protocol Class
- Internet Message Access Protocol Class
- Internet Server Class
- Mail Message Class
- Message Store Class
- Network News Protocol Class
- News Feed Class
- Post Office Protocol Class
- Remote Command Class
- Simple Mail Transfer Protocol Class
- Secure Shell Protocol Class
- SocketWrench Class
- Telnet Protocol Class
- Terminal Emulation Class
- Text Message Class
- Time Protocol Class
- Web Storage Class
- Whois Protocol Class

Domain Name Service Class Library

Resolve domain names into Internet addresses and return information about a remote host, such as the servers that are responsible for accepting mail for the domain.

Reference

- [Class Methods](#)
- [Constants](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

Class Name	CDnsClient
File Name	CSDNSV10.DLL
Version	10.0.1468.2518
LibID	F247C1C3-4DB3-4DF2-A16E-88E2B97B5119
Import Library	CSDNSV10.LIB
Dependencies	None
Standards	RFC 1034

Overview

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

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an

ActiveX component, and does not require COM registration.

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

Domain Name Service Class Methods

Class	Description
CDnsClient	Constructor which initializes the current instance of the class
~CDnsClient	Destructor which releases resources allocated by the class
Method	Description
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified client handle to another thread
Cancel	Cancel an outstanding nameserver query
DetachHandle	Detach the handle for the current instance of this class
DisableTrace	Disable logging of socket function calls to the trace log
EnableTrace	Enable logging of socket function calls to a file
EnumHostAliases	Enumerate the aliases for the specified host name or address
EnumMailExchanges	Return a list of mail exchanges for the specified host name or IP address
FormatAddress	Convert a numeric IPv4 address to a string
GetAddress	Convert an address string in dotted notation to a numeric IPv4 address
GetAddressFamily	Return the address family for the specified IP address
GetDefaultHostFile	Return the fully qualified path name of the default host file on the local system
GetErrorString	Return a description for the specified error code
GetHandle	Return the client handle used by this instance of the class
GetHostAddress	Return the IP address of the specified hostname
GetHostByAddress	Return a pointer to data for the specified host IP address
GetHostByName	Return a pointer to data for the specified host name
GetHostFile	Return the name of the current host file
GetHostInfo	Return additional information for the specified host
GetHostName	Return the host name for the specified IP address
GetHostServices	Return a list of services supported by the specified host
GetLastError	Return the last error code
GetLocalAddress	Return the IP address for the local host
GetLocalDomain	Return the local domain name for the current session
GetLocalName	Return the local host name
GetMailExchange	Return the host that processes mail for the specified domain
GetRecord	Return record data for the current host
GetResolverAddress	Return address of last nameserver that resolved query
GetResolverOptions	Return the current resolver options for the client session
GetRetryCount	Get the number of times the query is sent to each server

GetServerAddress	Return the address of the specified nameserver
GetServerPort	Return the port of the specified nameserver
GetTimeout	Get the number of seconds until a query times out
HostNameToUnicode	Converts the canonical form of a host name to its Unicode version
IsInitialized	Determine if the class has been successfully initialized
MatchHostName	Match a host name against of list of addresses including wildcards
NormalizeHostName	Return the canonical form of a host name
RegisterServer	Add a nameserver address to the current session
Reset	Reset the current client state
Resolve	Resolve a host name into an IP address or an IP address into a host name
SetHostFile	Specify the name of an alternate file to use when resolving host names and IP addresses
SetLastError	Set the last error code
SetLocalDomain	Set the local domain name for the current session
SetResolverOptions	Set the resolver options for the client session
SetRetryCount	Set the number of times the query is sent to each server
SetTimeout	Set the number of seconds until a query times out
ShowError	Display a message box with a description of the specified error
UnregisterServer	Remove a nameserver address from the current session

CDnsClient::CDnsClient Method

`CDnsClient();`

The **CDnsClient** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **DnsInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[~CDnsClient](#), [IsInitialized](#), [SetResolverOptions](#)

CDnsClient::~CDnsClient

`~CDnsClient();`

The **CDnsClient** destructor releases resources allocated by the current instance of the **CDnsClient** object. It also uninitialized the library if there are no other concurrent uses of the class.

Remarks

When a **CDnsClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CDnsClient](#)

CDnsClient::AttachHandle Method

```
VOID AttachHandle(  
    HCLIENT hClient  
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle already has been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the *hClient* parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

See Also

[DetachHandle](#), [GetHandle](#)

CDnsClient::AttachThread Method

```
DWORD AttachThread(  
    DWORD dwThreadId  
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the class. A value of zero specifies that the current thread should become the owner of the class instance.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is `DNS_ERROR`. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

See Also

[AttachHandle](#), [Cancel](#)

CDnsClient::Cancel Method

```
INT Cancel();
```

The **Cancel** method cancels any outstanding queries initiated by the client.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `DNS_ERROR`. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostByAddress](#), [GetHostByName](#), [GetHostInfo](#), [GetHostServices](#), [GetMailExchange](#), [GetRecord](#)

CDnsClient::DetachHandle Method

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

See Also

[AttachHandle](#), [GetHandle](#)

CDnsClient::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableTrace](#)

CDnsClient::EnableTrace Method

```
BOOL EnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **EnableTrace** method enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the remote host.

Trace method logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableTrace](#)

CDnsClient::EnumHostAliases Method

```
INT EnumHostAliases(  
    LPCTSTR lpszHostName,  
    LPTSTR *lpszHostAlias,  
    INT nMaxAliases  
);
```

The **EnumHostAliases** method returns a list of aliases for the specified host name or IP address.

Parameters

lpszHostName

Pointer to the string that contains the hostname to be resolved. If a fully qualified domain name is not provided, the default local domain will be used. This value may also be an IP address.

lpszHostAlias

Pointer to an array of string pointers which specify one or more host aliases. If the application needs to store these values, a local copy should be made because they are invalidated when another host name is resolved.

nMaxAliases

The maximum number of aliases in the array. This parameter must have a value of at least one, or an error will be returned.

Return Value

If the method succeeds, the return value is the number of host aliases. If the method fails, the return value is `DNS_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The application must never attempt to modify the host aliases or delete any of the values. This method uses an internal data structure to store the host information and only one copy of this structure is allocated per thread. The application must copy any information it needs before issuing any other function calls.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostAddress](#), [GetHostName](#)

CDnsClient::EnumMailExchanges Method

```
INT EnumMailExchanges(  
    LPCTSTR lpszHostName,  
    LPTSTR *lpszMailExchanges,  
    INT nMaxMailExchanges  
);
```

The **EnumMailExchanges** method returns a list of mail exchanges for the specified host name or IP address.

Parameters

lpszHostName

Pointer to the string that contains the hostname or domain name to be queried.

lpszMailExchanges

Pointer to an array of string pointers which specify one or more mail exchanges. If the application needs to store these values, a local copy should be made because they are invalidated when another host name is resolved. The list of mail exchange records is sorted in priority order, from highest (i.e., those whose preference value is smallest) to lowest.

nMaxMailExchanges

The maximum number of mail exchanges in the array. If this parameter is 0, then the method will return the number of mail exchanges, but the list of mail exchanges will not be output in *lpszMailExchanges*.

Return Value

If the method succeeds, the return value is the number of mail exchanges. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The application must never attempt to modify the mail exchange host names or delete any of the values. This method uses an internal data structure to store the host information and only one copy of this structure is allocated per thread. The application must copy any information it needs before issuing any other method calls.

Example

```
// Get the number of mail exchanges  
if ((nMX = pClient->EnumMailExchanges(szHostName, NULL, 0)) == DNS_ERROR)  
{  
    pClient->ShowError();  
}  
else  
{  
    // Allocate memory for the list of mail exchanges  
    lpszMailExchanges = (LPTSTR *)LocalAlloc(LPTR, (nMX * sizeof(LPTSTR)));  
    // Retrieve the list of mail exchanges  
    nMX = pClient->EnumMailExchanges(szHostName, lpszMailExchanges, nMX);  
  
    // Populate a listbox with the mail exchanges  
    for (int nIndex = 0; nIndex < nMX; nIndex++)  
        pListBox->AddString(*lpszMailExchanges++);  
  
    LocalFree((HLOCAL)lpszMailExchanges);  
}
```

}

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnav10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetMailExchange](#)

CDnsClient::FormatAddress Method

```
INT FormatAddress(  
    LPINTERNET_ADDRESS LpAddress,  
    LPTSTR LpszAddress,  
    INT nMaxLength  
);
```

```
INT FormatAddress(  
    LPINTERNET_ADDRESS LpAddress,  
    CString& strAddress  
);
```

The **FormatAddress** method converts a numeric IPv4 or IPv6 address to a string.

Parameters

lpAddress

A pointer to an INTERNET_ADDRESS structure which specifies the numeric IPv4 or IPv6 address to be converted into a string.

lpszAddress

A pointer to a null-terminated array of characters which will contain the converted IPv4 address in dot-notation. This string should be at least 16 characters in length. If the Microsoft Foundation Classes are being used, the second form of this method may be called where a **CString** object is specified instead.

nMaxLength

The maximum number of characters which may be copied in to the string buffer.

Return Value

If the method succeeds, the return value is the length of the string buffer. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetAddress](#), [GetHostAddress](#), [GetHostInfo](#), [GetHostServices](#), [GetMailExchange](#), [GetRecord](#), [INTERNET_ADDRESS](#)

CDnsClient::GetAddress Method

```
INT GetAddress(  
    LPCTSTR lpszAddress,  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddress  
);
```

The **GetAddress** method converts an address string to a numeric IPv4 or IPv6 address.

Parameters

lpszAddress

A pointer to a string which specifies an IPv4 address in dotted notation.

nAddressFamily

An integer value which specifies the type of IP address. If this parameter is zero, the address family will be determined automatically based on the format of the address string. If this parameter is DNS_ADDRESS_IPV4, the address must be in IPv4 format, and if it is DNS_ADDRESS_IPV6, the address must be in IPv6 format.

lpAddress

A pointer to an INTERNET_ADDRESS structure that will contain the numeric form of the IPv4 or IPv6 address in network byte order when the method returns.

Return Value

If the method succeeds, the return value is the address family for the IP address. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method will only accept a string that is in the proper format for an IP address, and cannot be used to resolve a host name. To perform host name resolution, use the **GetHostAddress** method. To convert a numeric address to an address string, use the **FormatAddress** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FormatAddress](#), [GetHostAddress](#), [GetHostInfo](#), [GetHostServices](#), [GetMailExchange](#), [GetRecord](#), [INTERNET_ADDRESS](#)

CDnsClient::GetAddressFamily Method

```
INT GetAddressFamily(  
    LPCTSTR lpszAddress  
);
```

The **GetAddressFamily** method returns the address family for the specified IP address.

Parameters

lpszAddress

A pointer to a string which specifies an IPv4 or IPv6 address.

Return Value

If the method succeeds, the return value is DNS_ADDRESS_IPV4 if the address is in IPv4 format, or DNS_ADDRESS_IPV6 if the address is in IPv6 format. If the address string is not in a recognized format, it returns DNS_ADDRESS_UNKNOWN.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetHostAddress](#), [DnsGetHostInfo](#), [DnsGetHostServices](#), [DnsGetMailExchange](#), [DnsGetRecord](#)

CDnsClient::GetDefaultHostFile Method

```
INT GetDefaultHostFile(  
    LPTSTR lpszFileName,  
    INT nMaxLength  
);
```

The **GetDefaultHostFile** method returns the fully qualified path name of the host file on the local system. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **GetHostAddress** and **GetHostName** method. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

```
ipaddress hostname [hostalias ...]
```

For example, one typical entry maps the name "localhost" to the local loopback IP address. This would be entered as:

```
127.0.0.1 localhost
```

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

The default hosts file is stored in C:\Windows\system32\drivers\etc\hosts and may or may not exist on a given system. Note that there is no extension for this file.

Parameters

lpszFileName

Pointer to a string buffer that will contain the fully qualified file name to the default host file. It is recommended that this buffer be at least MAX_PATH characters in size. This parameter may be NULL, in which case the method will return the length of the string, not including the terminating null byte.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the method succeeds, the return value is length of the string. A return value of zero indicates that the default host file could not be determined for the current platform. To get extended error information, call **GetLastError**.

Remarks

This method only returns the default location of the host file and does not determine if the file actually exists. It is not required that a host file be present on the system.

The default host file is processed before performing a nameserver lookup when resolving a hostname into an IP address, or an IP address into a hostname.

To specify an alternate local host file, use the **SetHostFile** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostAddress](#), [GetHostFile](#), [GetHostName](#), [SetHostFile](#)

CDnsClient::GetErrorString Method

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT nMaxLength  
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The last-error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is 0, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLastError](#), [SetLastError](#), [ShowError](#)

CDnsClient::GetHandle Method

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

See Also

[AttachHandle](#), [DetachHandle](#), [IsInitialized](#)

CDnsClient::GetHostAddress Method

```
INT GetHostAddress(  
    LPCTSTR lpszHostName,  
    INT nAddressFamily,  
    LPTSTR lpszHostAddress,  
    INT nMaxLength  
);
```

```
INT GetHostAddress(  
    LPCTSTR lpszHostName,  
    LPTSTR lpszHostAddress,  
    INT nMaxLength  
);
```

```
INT GetHostAddress(  
    LPCTSTR lpszHostName,  
    INT nAddressFamily,  
    CString& strHostAddress  
);
```

```
INT GetHostAddress(  
    LPCTSTR lpszHostName,  
    CString& strHostAddress  
);
```

The **GetHostAddress** method resolves the specified host name, storing the IP address in the provided buffer.

Parameters

lpszHostName

Pointer to the string that contains the hostname to be resolved. If a fully qualified domain name is not provided, the default local domain will be used.

nAddressFamily

An integer value which specifies the type of address that should be returned. A value of `DNS_ADDRESS_IPV4` specifies that the IPv4 address for the host should be returned. A value of `DNS_ADDRESS_IPV6` specifies that the IPv6 address for the host should be returned. A value of `DNS_ADDRESS_ANY` specifies that if the host only has an IPv6 address, that value should be returned, otherwise return the IPv4 address for the host. There are alternate versions of this method which omits the address family, in which case it will default to returning the IPv4 address for the host.

lpszHostAddress

Pointer to the buffer that will contain the IP address, stored as a string in dot notation. This buffer should be at least 48 characters in length. The format of the address is determined by the address family specified.

nMaxLength

The maximum length of the string buffer. The maximum length of the buffer must include the terminating null character.

Return Value

If the method succeeds, the return value is the number of characters copied into the host address buffer. If the method fails, the return value is `DNS_ERROR`. To get extended error information, call

GetLastError.

Remarks

The **GetHostAddress** method may return an address in either IPv4 or IPv6 format, depending on the address family that is specified and what records exist for the host. If your application does not support the IPv6 address format, you must specify the *nAddressFamily* parameter as `DNS_ADDRESS_IPV4` to prevent the possibility of an IPv6 address being returned.

If the *nAddressFamily* parameter is specified as `DNS_ADDRESS_ANY`, this function will first check for an IPv4 address record for the host. If it exists, it will return that address. If the host does not have an IPv4 address, it will then check for an IPv6 address record and return that address. This gives preference to IPv4 addresses, but your application should never depend on this behavior. In the future, this function may change to give preference to IPv6 addresses.

To determine what format an address is in, use the **GetAddressFamily** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnumHostAliases](#), [GetAddressFamily](#), [GetHostName](#), [GetRecord](#)

CDnsClient::GetHostByAddress Method

```
LPHOSTENT GetHostByAddress(  
    LPVOID LpvAddress,  
    INT cbAddress,  
    INT nAddressFamily  
);
```

The **GetHostByAddress** method returns a pointer to a [HOSTENT](#) structure which contains the results of a successful search for the host specified by address parameter.

Parameters

lpvAddress

Pointer to an integer IPv4 address in network byte order.

cbAddress

The length of the address in bytes; this value should always be 4.

nAddressFamily

The type of address being resolved; this value should always be `DNS_ADDRESS_IPV4` as defined in the Windows Sockets header file.

Return Value

If the method succeeds, the return value is a pointer to a `HOSTENT` structure. If the method fails, the return value is `NULL`. To get extended error information, call **GetLastError**.

Remarks

The application must never attempt to modify this structure or to free any of its components. Only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other function calls. To convert an IPv4 address string in dotted notation to a 32-bit IP address, use the **GetAddress** method.

This method is included for compatibility with existing applications which already use the `HOSTENT` structure. Because this method returns a pointer to a complex structure, it may not be suitable for some programming languages.

This method is not compatible with IPv6 addresses. For applications that must support both IPv4 and IPv6 address formats, use the **GetHostAddress** and **GetHostName** methods.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostAddress](#), [GetHostByName](#), [GetHostInfo](#), [GetHostName](#), [GetHostServices](#), [GetRecord](#), [GetResolverAddress](#), [RegisterServer](#)

CDnsClient::GetHostByName Method

```
LPHOSTENT GetHostByName(  
    LPCTSTR lpszHostName  
);
```

The **GetHostByName** method returns a pointer to a [HOSTENT](#) structure which contains the results of a successful search for the host specified in the name parameter.

Parameters

lpszHostName

Pointer to the string that contains the hostname to be resolved. If a fully qualified domain name is not provided, the default local domain will be used.

Return Value

If the method succeeds, the return value is a pointer to a [HOSTENT](#) structure. If the method fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The application must never attempt to modify this structure or to free any of its components. Only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other function calls. This method will automatically resolve an IP address passed as a string, converting it to numeric form and calling the **GetHostByAddress** method.

This method is included for compatibility with existing applications which already use the [HOSTENT](#) structure. Because this method returns a pointer to a complex structure, it may not be suitable for some programming languages.

This method is not compatible with IPv6 addresses. For applications that must support both IPv4 and IPv6 address formats, use the **GetHostAddress** and **GetHostName** methods.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostAddress](#), [GetHostByAddress](#), [GetHostInfo](#), [GetHostName](#), [GetHostServices](#),
[GetMailExchange](#), [GetRecord](#), [GetResolverAddress](#), [RegisterServer](#)

CDnsClient::GetHostFile Method

```
INT GetHostFile(  
    LPTSTR lpszFileName,  
    INT nMaxLength  
);  
  
INT GetHostFile(  
    CString& strFileName  
);
```

The **GetHostFile** method returns the name of the host file previously set using the **SetHostFile** method. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **GetHostAddress** and **GetHostName** method.

Parameters

lpszFileName

Pointer to a string buffer that will contain the host file name. It is recommended that this buffer be at least MAX_PATH characters in size. This parameter may be NULL, in which case the method will return the length of the string, not including the terminating null character.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the method succeeds, the return value is length of the string. A return value of zero indicates that no host file has been specified or the method was unable to determine the file name. To get extended error information, call **GetLastError**. If the last error is zero, this indicates that no host file name has been specified for the current thread. If the last error is non-zero, this indicates the reason that the method failed.

Remarks

This method only returns the name of the host file that is cached in memory for the current thread. The contents of the file on the disk may have changed after the file was loaded into memory. To reload the host file or clear the cache, call the **SetHostFile** method.

If a host file has been specified, it is processed before the default host file when resolving a hostname into an IP address, or an IP address into a hostname. If the host name or address is not found, or no host file has been specified, a nameserver lookup is performed.

The host file returned by this method may be different than the default host file for the local system. To determine the file name for the default host file, use the **GetDefaultHostFile** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetDefaultHostFile](#), [GetHostAddress](#), [GetHostName](#), [SetHostFile](#)

CDnsClient::GetHostInfo Method

```
INT GetHostInfo(  
    LPCTSTR lpszHostName,  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);
```

The **GetHostInfo** method returns the HINFO record for the specified hostname. This information, if it is provided, typically specifies the operating system type and hardware platform.

Parameters

lpszHostName

Pointer to the string which specifies the host name that information will be returned for.

lpszBuffer

Pointer to the buffer which will contain the host information returned by the nameserver.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the method succeeds, the length of the host information buffer is returned. A return value of zero indicates that no information is available for the specified host. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Remarks

Many systems do not maintain HINFO records for a site since that information can potentially be used to compromise system security. The information is typically used for administrative purposes with internal networks.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostByAddress](#), [GetHostByName](#), [GetHostServices](#), [GetMailExchange](#), [GetRecord](#), [GetResolverAddress](#), [RegisterServer](#)

CDnsClient::GetHostName Method

```
INT GetHostName(  
    LPCTSTR lpszHostAddress,  
    LPTSTR lpszHostName,  
    INT nMaxLength  
);
```

```
INT GetHostName(  
    LPCTSTR lpszHostAddress,  
    CString& strHostName  
);
```

The **GetHostName** method resolves the specified IP address, storing the fully qualified host name in the provided buffer.

Parameters

lpszHostAddress

Pointer to a string that specifies an IPv4 or IPv6 formatted address.

lpszHostName

Pointer to the buffer that will contain the fully qualified domain name for the specified host. This buffer should be at least 64 characters in length.

nMaxLength

The maximum length of the string buffer.

Return Value

If the method succeeds, the return value is the number of characters copied into the host name buffer. If the method fails, the return value is `DNS_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **GetHostName** method first looks to see if there is an entry in the local host file for the specified IP address, and if one exists, it will return the host name for that address. If you do not want to use the local host file at all, and only return an host name if a DNS query resolves the address, use the **GetRecord** method and specify a record type of `DNS_RECORD_PTR`.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnumHostAliases](#), [GetHostByAddress](#), [GetHostByName](#), [GetHostAddress](#)

CDnsClient::GetHostServices Method

```
INT GetHostServices(  
    LPCTSTR lpszHostName,  
    INT nProtocol,  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);
```

The **GetHostServices** method returns the WKS (Well Known Services) record for the specified hostname and protocol. This information, if it is provided, typically specifies the names of those services supported on the host.

Parameters

lpszHostName

Pointer to the string which specifies the host name that information will be returned for.

nProtocol

The protocol for those services that information should be returned about. The following protocols are recognized:

Value	Constant	Description
6	DNS_PROTOCOL_TCP	Services that use the Transmission Control Protocol (TCP)
17	DNS_PROTOCOL_UDP	Services that use the User Datagram Protocol (UDP)

lpszBuffer

Pointer to the buffer which will contain the host information returned by the nameserver.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the method succeeds, the length of the host services buffer is returned. A return value of zero indicates that no information is available for the specified host. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Remarks

Many systems do not maintain complete services records for a site since that information can potentially be used to compromise system security. An application should not depend on this information being available for any given record.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostByAddress](#), [GetHostByName](#), [GetHostInfo](#), [GetMailExchange](#), [GetRecord](#),

CDnsClient::GetLastError Method

```
DWORD GetLastError();  
  
DWORD GetLastError(  
    CString& strDescription  
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError(0)** when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or DNS_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

See Also

[GetErrorString](#), [SetLastError](#), [ShowError](#)

CDnsClient::GetLocalAddress Method

```
INT GetLocalAddress(  
    INT nAddressFamily,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

The **DnsGetLocalAddress** method returns the IP address for the local host.

nAddressFamily

An integer value which specifies the type of address that should be returned. A value of DNS_ADDRESS_IPV4 specifies that the IPv4 address for the host should be returned. A value of DNS_ADDRESS_IPV6 specifies that the IPv6 address for the host should be returned. A value of DNS_ADDRESS_ANY specifies that if the host only has an IPv6 address, that value should be returned, otherwise return the IPv4 address for the host.

lpszAddress

Pointer to the buffer that will contain the IP address, stored as a string in dot notation. This buffer should be at least 40 characters in length to accommodate both IPv4 and IPv6 addresses.

nMaxLength

The maximum length of the string buffer.

Return Value

If the method succeeds, the return value is the number of characters copied into the host address buffer. If the method fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The **GetLocalAddress** method may return an address in either IPv4 or IPv6 format, depending on the address family that is specified and what records exist for the host. If your application does not support the IPv6 address format, you must specify the *nAddressFamily* parameter as DNS_ADDRESS_IPV4 to prevent the possibility of an IPv6 address being returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostName](#), [GetHostInfo](#), [GetHostServices](#), [GetMailExchange](#), [GetRecord](#)

CDnsClient::GetLocalDomain Method

```
INT GetLocalDomain(  
    LPTSTR lpszDomain,  
    INT nMaxLength  
);
```

The **GetLocalDomain** method copies the local domain name into the specified buffer. The value returned is the same value that was set with the **SetLocalDomain** method. If no local domain name has been set, an empty string is returned.

Parameters

lpszDomain

Pointer to the buffer that is used to store the local domain name. If no local domain name has been set, this buffer will be set to zero length.

nMaxLength

The maximum number of bytes to copy into the buffer, including the null character terminator.

Return Value

If the method succeeds, the return value is the length of the domain name string. A return value of zero indicates that no local domain name has been set. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RegisterServer](#), [SetLocalDomain](#)

CDnsClient::GetLocalName Method

```
INT GetLocalName(  
    LPTSTR lpszLocalName,  
    INT nMaxLength  
);
```

The **GetLocalName** method returns the local host name.

Parameters

lpszLocalName

Pointer to a string buffer that will contain the local host name. It is recommended that this buffer be at least 64 characters in size.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the method succeeds, the return value is the length of the local host name. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostByAddress](#), [GetHostByName](#), [GetHostInfo](#), [GetHostServices](#), [GetMailExchange](#), [GetRecord](#)

CDnsClient::GetMailExchange Method

```
INT GetMailExchange(  
    LPCTSTR LpszHostName,  
    LPINT LpnPreference,  
    LPTSTR LpszBuffer,  
    INT nMaxLength  
);
```

The **GetMailExchange** method returns the mail exchange (MX) record information for the specified domain. This information, if it is provided, identifies a server responsible for processing mail for the given domain.

Parameters

LpszHostName

Pointer to the string which specifies the host name that information will be returned for.

LpnPreference

Pointer to the integer which will contain the preference for the specified mail exchange host.

LpszBuffer

Pointer to the buffer which will contain the host information returned by the nameserver.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the method succeeds, the length of the buffer is returned. A return value of zero indicates that no information is available for the specified host. If the method fails, the return value is `DNS_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The mail exchange record is typically used by mail delivery agents to determine what system is responsible for accepting mail addressed to a given domain. This method will return the first MX record provided by the server. Note that some domains may have multiple mail servers. To enumerate all of the mail exchange records for a domain, use the **EnumMailExchanges** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostByAddress](#), [GetHostByName](#), [GetHostInfo](#), [GetHostServices](#), [GetRecord](#), [GetResolverAddress](#), [RegisterServer](#), [EnumMailExchanges](#)

CDnsClient::GetRecord Method

```
INT GetRecord(  
    LPCTSTR LpszHostName,  
    INT nRecordType,  
    LPTSTR LpszBuffer,  
    INT nMaxLength  
);
```

```
INT GetRecord(  
    LPCTSTR LpszHostName,  
    INT nRecordType,  
    CString& strBuffer  
);
```

The **GetRecord** method returns the specified record information for the given hostname.

Parameters

LpszHostName

Pointer to the string which specifies the host name that information will be returned for.

nRecordType

The record type for the information that should be returned. The following record types are recognized:

Value	Constant	Description
0	DNS_RECORD_NONE	No record type
1	DNS_RECORD_ADDRESS	Host address
2	DNS_RECORD_NS	Authoritative nameserver
5	DNS_RECORD_CNAME	Canonical name (alias)
6	DNS_RECORD_SOA	Start of Authority
11	DNS_RECORD_WKS	Well known services
12	DNS_RECORD_PTR	Domain name
13	DNS_RECORD_HINFO	Host information
14	DNS_RECORD_MINFO	Mailbox information
15	DNS_RECORD_MX	Mail exchange host
16	DNS_RECORD_TXT	Text strings
29	DNS_RECORD_LOC	Location information
100	DNS_RECORD_UINFO	User information
101	DNS_RECORD_UID	User ID
102	DNS_RECORD_GID	Group ID

LpszBuffer

Pointer to the buffer which will contain the host information returned by the nameserver.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the method succeeds, the length of the information buffer is returned. A return value of zero indicates that no information for that record is available for the specified host. If the method fails, the return value is `DNS_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **GetRecord** method can be used to resolve a host name into an IP address using the record type `DNS_RECORD_ADDRESS`. It can also be used to perform a reverse lookup and resolve an IP address into a host name by using the record type `DNS_RECORD_PTR`.

To determine the host that serves as the primary or master DNS for a zone, the record name should be specified as the domain name (e.g.: `microsoft.com`) and the record type should be `DNS_RECORD_SOA`. The value returned will be the fully qualified domain name for host.

Note that this method does not reference a local host file when resolving host names or addresses. If the record lookup fails, this method will return an error even if there's an entry for the host in the file that has been specified by a call to **SetHostFile**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostAddress](#), [GetHostName](#), [GetHostInfo](#), [GetHostServices](#), [GetMailExchange](#), [GetResolverAddress](#), [RegisterServer](#)

CDnsClient::GetResolverAddress Method

```
INT GetResolverAddress(  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);  
  
INT GetResolverAddress(  
    CString& strAddress  
);
```

The **GetResolverAddress** returns the address of the nameserver that resolved the last query.

Parameters

lpszAddress

A pointer to a string buffer that will contain the address of the nameserver when the function returns. This buffer should be large enough to store both IPv4 and IPv6 addresses, with a minimum length of 40 characters. If this parameter is NULL, it will be ignored.

nMaxLength

The maximum number of characters that can be copied into the string buffer. If this value is zero, the *lpszAddress* parameter will be ignored and the function will return the length of the address.

Return Value

If the method succeeds, the return value is the length of the address, not including the terminating null character. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostByAddress](#), [GetHostByName](#), [GetHostInfo](#), [GetHostServices](#), [GetMailExchange](#), [GetRecord](#), [RegisterServer](#)

CDnsClient::GetResolverOptions Method

```
DWORD GetResolverOptions();
```

The **GetResolverOptions** method returns the options that have been set for the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the resolver options set for the client session. If the client handle is invalid or no resolver options have been specified, the function will return zero.

Remarks

The **GetResolverOptions** method can be used to determine which resolver options have been specified for the client session. For a list of the available options, refer to the documentation for the **SetResolverOptions** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

See Also

[CDnsClient](#), [SetResolverOptions](#)

CDnsClient::GetRetryCount Method

```
INT GetRetryCount();
```

The **GetRetryCount** returns the retry count for the current client session.

Return Value

If the method succeeds, the return value is the retry count. If the method fails, the return value is `DNS_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The retry count determines the amount of time the client will wait for a response from each query, the effective amount of time the client will wait increases with each nameserver and the total number of retries specified. For example, two nameservers registered with the client, with a default of 4 retries per nameserver and a timeout value of 10 seconds, would cause the client to wait a total of 80 seconds until it returns an error indicating that it was unable to resolve the query.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetTimeout](#), [SetRetryCount](#), [SetTimeout](#)

CDnsClient::GetServerAddress Method

```
INT GetServerAddress(  
    INT nServer,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

```
INT GetServerAddress(  
    INT nServer,  
    CString& strAddress  
);
```

The **GetServerAddress** method returns the address of the specified nameserver.

Parameters

nServer

The index into the client's nameserver table. This index is the same value that is passed to the **RegisterServer** method when the nameserver is registered.

lpszAddress

A pointer to a string buffer that will contain the address of the nameserver when the function returns. This buffer should be large enough to store both IPv4 and IPv6 addresses, with a minimum length of 40 characters. If this parameter is NULL, it will be ignored.

nMaxLength

The maximum number of characters that can be copied into the string buffer. If this value is zero, the *lpszAddress* parameter will be ignored and the function will return the length of the address.

Return Value

If the method succeeds, the return value is the length of the address, not including the terminating null character. If the method fails, the return value is `DNS_ERROR`. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetResolverAddress](#), [GetServerPort](#), [RegisterServer](#), [UnregisterServer](#)

CDnsClient::GetServerPort Method

```
INT GetServerPort(  
    INT nServer  
);
```

The **GetServerPort** method returns the port number registered to the specified nameserver.

Parameters

nServer

The index into the client's nameserver table. This index is the same value that is passed to the **RegisterServer** method when the nameserver is registered.

Return Value

If the method succeeds, the return value is the specified port number. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetResolverAddress](#), [GetServerAddress](#), [RegisterServer](#), [UnregisterServer](#)

CDnsClient::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the timeout value for the current client session.

Return Value

If the method succeeds, the return value is the timeout period. If the method fails, the return value is `DNS_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The timeout value determines the amount of time the client will wait for a response from each query, the effective amount of time the client will wait increases with each nameserver and the total number of retries specified. For example, two nameservers registered with the client, with a default of 4 retries per nameserver and a timeout value of 10 seconds, would cause the client to wait a total of 80 seconds until it returns an error indicating that it was unable to resolve the query.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

See Also

[GetRetryCount](#), [SetRetryCount](#), [SetTimeout](#)

CDnsClient::HostNameToUnicode Method

```
INT HostNameToUnicode(  
    LPCTSTR lpszHostName,  
    LPTSTR lpszUnicodeName,  
    INT nMaxLength  
);  
  
INT HostNameToUnicode(  
    LPCTSTR lpszHostName,  
    CString& strUnicodeName  
);
```

The **HostNameToUnicode** method converts the canonical form of a host name to its Unicode version.

Parameters

lpszHostName

Pointer to the host name as a null-terminated string. This parameter cannot be a NULL pointer or a zero length string.

lpszUnicodeName

Pointer to the string buffer that will contain the original Unicode version of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer. An alternate version of this method accepts a reference to a CString object.

nMaxLength

The maximum number of characters that can be copied to the *lpszUnicodeName* string buffer. This parameter cannot be zero, and must include the terminating null character.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **HostNameToUnicode** method will convert the encoded ASCII version of a host name to its Unicode version. Although any valid host name is accepted by this method, it is intended to convert a Punycode encoded host name to its original Unicode character encoding.

If the application is compiled using the Unicode character set, the value returned in *lpszUnicodeName* will be a Unicode string using UTF-16 encoding. If the ANSI character set is used, the value returned will be a Unicode string using UTF-8 encoding. To display a UTF-8 encoded host name, your application will need to convert it to UTF-16 using the **MultiByteToWideChar** function.

Although this method performs checks to ensure that the *lpszHostName* parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the existence of the domain name. To check if the host name exists and has a valid IP address, use the [GetHostAddress](#) method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostAddress](#), [NormalizeHostName](#)

CDnsClient::IsInitialized Method

BOOL IsInitialized();

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csdnsv10.lib

See Also

[CDnsClient](#)

CDnsClient::MatchHostName Method

```
BOOL MatchHostName(  
    LPCTSTR lpszHostName,  
    LPCTSTR lpszHostMask  
    BOOL bResolve  
);
```

The **MatchHostName** method matches a host name against one or more strings that may contain wildcards.

Parameters

lpszHostName

A pointer to a string which specifies the host name or IP address to match.

lpszHostMask

A pointer to a string which specifies one or more values to match against the host name. The asterisk character can be used to match any number of characters in the host name, and the question mark can be used to match any single character. Multiple values may be specified by separating them with a semicolon.

bResolve

A boolean value which specifies if the host name or IP address should be resolved when matching the host against the mask string. If this parameter is non-zero, two checks against the host mask string will be performed; once for the host name specified and once for its IP address. If this parameter is zero, then the match is made only against the host name string provided.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **MatchHostName** method provides a convenient way for an application to determine if a given host name matches one or more mask strings which may contain wildcard characters. For example, the host name could be "www.microsoft.com" and the host mask string could be "*.microsoft.com". In this example, the method would return a non-zero value indicating the host name matched the mask. However, if the mask string was "*.net" then the method would return zero, indicating that there was no match. Multiple mask values can be combined by separating them with a semicolon; for example, the mask "*.com;*.org" would match any host name in either the .com or .org top-level domains.

If an internationalized domain name (IDN) is specified, it will be converted internally to an ASCII string using Punycode encoding. The host mask will be matched against this encoded version of the host name, not its Unicode version.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetAddress](#), [GetHostAddress](#), [GetHostName](#), [GetLocalAddress](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CDnsClient::NormalizeHostName Method

```
INT NormalizeHostName(  
    LPCTSTR lpszHostName,  
    LPTSTR lpszNormalized,  
    INT nMaxLength  
);
```

```
INT NormalizeHostName(  
    LPCTSTR lpszHostName,  
    CString& strNormalized  
);
```

The **NormalizeHostName** method returns the canonical form of a host name in the specified buffer.

Parameters

lpszHostName

Pointer to the host name as a null-terminated string. This parameter cannot be a NULL pointer or a zero length string.

lpszNormalized

Pointer to the string buffer that will contain the canonical form of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer. An alternate version of this method accepts a reference to a CString object.

nMaxLength

The maximum number of characters that can be copied to the *lpszNormalized* string buffer. This parameter cannot be zero, and must include the terminating null character.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **NormalizeHostName** method will remove all leading and trailing whitespace characters from the host name and fold all upper-case characters to lower-case. If an internationalized domain name (IDN) containing Unicode characters is passed to this method, it will be converted to an ASCII compatible format for domain names.

If the application is compiled using the Unicode character set, the host name will be converted from UTF-16 to UTF-8 and then processed. If you are unsure if an internationalized domain name will be specified as the host name, it is recommended that you use Unicode.

Although this method performs checks to ensure that the *lpszHostName* parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the existence of the domain name. To check if the host name exists and has a valid IP address, use the [GetHostAddress](#) method.

It is recommended that you use this method if your application needs to store the host name, and if accepts a host name as user input. It is not necessary to call this method prior to calling the other methods that accept a host name as a parameter. They already normalize the host name

and perform checks to ensure it is in the correct format.

If the *lpzHostName* parameter specifies a valid IPv4 or IPv6 address string instead of a host name, this method will return a copy of that IP address in the buffer provided by the caller. This allows the method to be used in cases where a user may input either a host name or IP address. To determine if the IP address has a corresponding host name, use the [GetHostName](#) method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostAddress](#), [GetHostName](#), [HostNameToUnicode](#)

CDnsClient::RegisterServer Method

```
INT RegisterServer(  
    INT nServer,  
    LPCTSTR lpszHostAddress,  
    INT nPort  
);
```

The **RegisterServer** method registers a nameserver with the current client session. The nameserver is used to resolve queries issued by the client, such as returning the IP address for a given host name. At least one nameserver must be registered by the client before queries are issued.

Parameters

nServer

The index into the client nameserver table. This index, starting at 0, is used to specify which slot in the client's nameserver table will be used to store the nameserver information.

lpszHostAddress

A pointer to a string that specifies the IP address of the nameserver to be registered. Note that hostnames cannot be specified.

nPort

The port number that the specified nameserver is accepting queries on. This value may be set to zero, in which case it will use the default port value.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `DNS_ERROR`. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetServerAddress](#), [GetServerPort](#), [UnregisterServer](#)

CDnsClient::Reset Method

```
INT Reset();
```

The **Reset** method resets the current state of the client session. The timeout and retry counts are set to their default values, the local domain name is cleared and all registered servers are removed from the client nameserver table.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `DNS_ERROR`. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RegisterServer](#), [SetLocalDomain](#), [SetRetryCount](#), [SetTimeout](#), [UnregisterServer](#)

CDnsClient::Resolve Method

```
BOOL Resolve(  
    LPCTSTR lpszRecord,  
    LPTSTR lpszResult,  
    INT nMaxLength,  
    BOOL bHostName  
);  
  
BOOL Resolve(  
    LPCTSTR lpszRecord,  
    LPINTERNET_ADDRESS lpAddress  
);  
  
BOOL Resolve(  
    LPINTERNET_ADDRESS lpAddress,  
    LPTSTR lpszHostName,  
    INT nMaxLength  
);  
  
BOOL Resolve(  
    LPCTSTR lpszRecord,  
    CString& strResult,  
    BOOL bHostName  
);  
  
BOOL Resolve(  
    LPINTERNET_ADDRESS lpAddress,  
    CString& strHostName  
);
```

The **Resolve** method resolves the specified host name into an IP address, or an IP address into its corresponding host name.

Parameters

lpszRecord

Pointer to a string which specifies the record to be resolved. If the *bHostName* argument is non-zero, then the record is expected to be a host name; otherwise it is expected to be an IP address string in dot notation.

lpszResult

Pointer to the buffer that will contain the result of the nameserver query. If the *bHostName* argument is non-zero, then the string will contain the host's IP address in dot notation when the method returns; otherwise, it will contain the host name for the IP address specified in the *lpszRecord* argument.

nMaxLength

The maximum number of characters that can be copied into the string buffer.

bHostName

A boolean argument which determines how the record is resolved. If this value is non-zero, then the *lpszRecord* argument should specify a host name and the *lpszResult* buffer will contain the IP address for that host when the method returns. If this value is zero, then the *lpszRecord* argument should specify an IP address and the *lpszResult* buffer will contain its host name when the method returns.

lpAddress

A pointer to an `INTERNET_ADDRESS` structure which specifies the numeric form of an IPv4 or IPv6 address in network byte order. This can be used with versions of the method that resolve host names and addresses using the numeric form of the IP address instead of a string.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Resolve** method provides a convenient interface to resolve host names and IP addresses. Note that this method first looks to see if there is an entry in the local host file for the specified host name or IP address, and if one exists, it will return that record without querying the name server.

Example

```
// Create an instance of the class object
CDnsClient dnsClient;

// Resolve the address of the www.microsoft.com server
CString strHostName = _T("www.microsoft.com");
INTERNET_ADDRESS ipHostAddress;

// If the host name is resolved, then perform a reverse
// lookup on that address to obtain the actual host name
// for that server
if (dnsClient.Resolve(strHostName, &ipHostAddress))
    dnsClient.Resolve(&ipHostAddress, strHostName);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetAddress](#), [FormatAddress](#), [GetHostAddress](#), [GetHostFile](#), [GetHostName](#), [SetHostFile](#), [INTERNET_ADDRESS](#)

CDnsClient::SetHostFile Method

```
INT SetHostFile(  
    LPCTSTR lpszFileName  
);
```

The **SetHostFile** method specifies the name of an alternate file to use when resolving hostnames and IP addresses. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **GetHostAddress** and **GetHostName** methods. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

```
ipaddress hostname [hostalias ...]
```

For example, one typical entry maps the name "localhost" to the local loopback IP address. This would be entered as:

```
127.0.0.1 localhost
```

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

Parameters

lpszFileName

Pointer to a string that specifies the name of the file. If the parameter is NULL, then the current host file is cleared from the cache and only the default host file will be used to resolve hostnames and addresses.

Return Value

If the method succeeds, the return value is the number of entries in the host file. A return value of DNS_ERROR indicates failure. To get extended error information, call **GetLastError**.

Remarks

This method loads the file into memory allocated for the current thread. If the contents of the file have changed after the method has been called, those changes will not be reflected when resolving hostnames or addresses. To reload the host file from disk, call this method again with the same file name. To remove the alternate host file from memory, specify a NULL pointer as the parameter.

If a host file has been specified, it is processed before the default host file when resolving a hostname into an IP address, or an IP address into a hostname. If the host name or address is not found, or no host file has been specified, a nameserver lookup is performed.

To determine if an alternate host file has been specified, use the **GetHostFile** method. A return value of zero indicates that no alternate host file has been cached for the current thread.

A system may have a default host file, which is used to resolve hostnames before performing a nameserver lookup. To determine the name of this file, use the **GetDefaultHostFile** method. It is not necessary to specify this default host file, since it is always used to resolve host names and addresses.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetDefaultHostFile](#), [GetHostAddress](#), [GetHostFile](#), [GetHostName](#)

CDnsClient::SetLastError Method

```
VOID SetLastError(  
    DWORD dwErrorCode  
);
```

The **SetLastError** method sets the error code for the current thread. This method is typically used to clear the last error by passing a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the error code for the current thread. A value of zero clears the last error code.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or DNS_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

See Also

[GetErrorString](#), [GetLastError](#), [ShowError](#)

CDnsClient::SetLocalDomain Method

```
INT SetLocalDomain(  
    LPCTSTR lpszDomain  
);
```

Parameters

lpszDomain

Pointer to the string which contains the local domain name. This is used as a default value when a query does not explicitly specify a domain name.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLocalDomain](#), [GetRetryCount](#), [GetTimeout](#), [Reset](#), [SetRetryCount](#), [SetTimeout](#)

CDnsClient::SetResolverOptions Method

```
DWORD SetResolverOptions(  
    DWORD dwOptions  
);
```

The **SetResolverOptions** method changes the resolver options for the client session.

Parameters

hClient

Handle to the client session.

dwOptions

An unsigned integer that specifies one or more resolver options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
DNS_OPTION_NONE	No additional resolver options specified. This is the default value, and it is recommended that most applications do not specify additional options unless the implications of doing so are understood.
DNS_OPTION_PROMOTE	Promotes the server that successfully completed the last query to the first server that will be used to resolve subsequent queries. This option can improve performance in some cases where one or more of the registered servers are non-responsive. This option takes precedence over the DNS_OPTION_ROTATE option.
DNS_OPTION_ROTATE	Enables a round-robin selection of nameservers when performing queries. Normally each nameserver is queried in the same order. This option rotates the available nameservers so a different server is used with each query.
DNS_OPTION_AUTHONLY	Require the answer from the nameserver to be authoritative, not from the server's cache. This option is included for future expansion as most servers do not support this feature and will ignore it.
DNS_OPTION_PRIMARY	Queries are only accepted from the primary nameserver. This option is included for future expansion as most servers do not support this feature and will ignore it.
DNS_OPTION_NORECURSE	Disable the sending of recursive queries to the nameserver. Specifying this option will disable the bit in the DNS request header that specifies recursion is desired.
DNS_OPTION_NOSEARCH	Disable additional queries of higher domains in the search list if the host name cannot be resolved. If this option is specified, and the host name cannot be

	resolved using the local domain name an error is returned immediately. This option is ignored if no local domain has been specified or if the DNS_OPTION_NOSUFFIX option has been specified.
DNS_OPTION_NOSUFFIX	Disable additional queries using the local domain name if the host name is not a fully qualified domain name and cannot be resolved. This option is ignored if no local domain has been specified.
DNS_OPTION_NONAMECHECK	Disable checking the host name for invalid characters, such as the underscore and control characters. By default, host names are checked to ensure they're valid before submitting a query to the nameserver.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetResolverOptions** method changes the resolver options for the specified client session, modifying how nameserver queries are processed. It is recommended that most applications do not specify any resolver options and use the default behavior. Specifying these options without understanding how they can affect standard queries can result in unexpected failures. In particular, caution should be used when specifying the DNS_OPTION_NORECURSE and DNS_OPTION_NOSEARCH options as they change the normal process of resolving a host name.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csdnsv10.lib

See Also

[CDnsClient](#), [GetResolverOptions](#)

CDnsClient::SetRetryCount Method

```
INT SetRetryCount(  
    INT nRetries  
);
```

The **SetRetryCount** method sets the number of attempts that the client will make attempting to resolve a query. When used in conjunction with the **SetTimeout** method, it determines the total amount of time the client will spend attempting to resolve a query.

Parameters

nRetries

The number of attempts the client will make, per nameserver, to resolve a query.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The retry count determines the amount of time the client will wait for a response from each query, the effective amount of time the client will wait increases with each nameserver and the total number of retries specified. For example, two nameservers registered with the client, with a default of 4 retries per nameserver and a timeout value of 10 seconds, would cause the client to wait a total of 80 seconds until it returns an error indicating that it was unable to resolve the query.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetRetryCount](#), [GetTimeout](#), [SetTimeout](#)

CDnsClient::SetTimeout Method

```
INT SetTimeout(  
    INT nTimeout  
);
```

The **SetTimeout** method sets the number of seconds that the client will wait for a response from a nameserver. The timeout value is used each time a server in the client's nameserver table is queried. When used in conjunction with the **SetRetryCount** method, it determines the total amount of time the client will spend attempting to resolve a query.

Parameters

nTimeout

The number of seconds until the client times out waiting for a response from a nameserver.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value determines the amount of time the client will wait for a response from each query, the effective amount of time the client will wait increases with each nameserver and the total number of retries specified. For example, with two nameservers registered with the client, with a default of 4 retries per nameserver and a timeout value of 10 seconds, would cause the client to wait a total of 80 seconds until it returns an error indicating that it was unable to resolve the query.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csdnsv10.lib

See Also

[GetRetryCount](#), [GetTimeout](#), [SetRetryCount](#)

CDnsClient::ShowError Method

```
INT ShowError(  
    LPCTSTR lpszAppTitle,  
    UINT uType,  
    DWORD dwErrorCode  
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A pointer to a string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** function in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that function for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetErrorString](#), [GetLastError](#), [SetLastError](#)

CDnsClient::UnregisterServer Method

```
INT UnregisterServer(  
    INT nServer  
);
```

The **UnregisterServer** method removes the specified nameserver information from the client. Unregistering a server prevents the client from using that server to satisfy subsequent DNS queries.

Parameters

nServer

The index into the client's nameserver table. This index is the same value that is passed to the **RegisterServer** method when the nameserver is registered.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is DNS_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetServerAddress](#), [GetServerPort](#), [RegisterServer](#), [Reset](#)

Domain Name Service Data Structures

- HOSTENT
- INTERNET_ADDRESS

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

HOSTENT Structure

This structure is used by the [GetHostByAddress](#) and [GetHostByName](#) methods to return information about a specified host. The application must never attempt to modify this structure or to free any of its components.

Only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other function calls. This is the same data structure used by the Windows Sockets API.

```
typedef struct _HOSTENT
{
    char *   h_name;
    char **  h_aliases;
    short    h_addrtype;
    short    h_length;
    char **  h_addr_list;
} HOSTENT, *LPHOSTENT;
```

Members

h_name

The fully qualified domain name (FQDN) that caused the nameserver server to return a reply.

h_aliases

A NULL-terminated array of alternate names.

h_addrtype

The type of address being returned.

h_length

The length, in bytes, of each address.

h_addr_list

A NULL-terminated list of addresses for the host. Addresses are returned in network byte order. The macro **h_addr** is defined to be **h_addr_list[0]** for compatibility with older software.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include winsock.h.

INTERNET_ADDRESS Structure

This structure represents a numeric IPv4 or IPv6 address in network byte order.

```
typedef struct _INTERNET_ADDRESS
{
    INT    ipFamily;
    BYTE   ipNumber[16];
} INTERNET_ADDRESS, *LPINTERNET_ADDRESS;
```

Members

ipFamily

An integer which identifies the type of IP address. It will be one of the following values:

Constant	Description
INET_ADDRESS_UNKNOWN	The address has not been specified or the bytes in the <i>ipNumber</i> array does not represent a valid address. Functions which populate this structure will use this value to indicate that the address cannot be determined.
INET_ADDRESS_IPV4	Specifies that the address is in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address is in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

ipNumber

A byte array which contains the numeric form of the IP address. This array is large enough to store both IPv4 (32 bit) and IPv6 (128 bit) addresses. The values are stored in network byte order.

Remarks

The **INTERNET_ADDRESS** structure is used by some functions to represent an Internet address in a binary format that is compatible with both IPv4 and IPv6 addresses. Applications that use this structure should consider it to be opaque, and should not modify the contents of the structure directly.

For compatibility with legacy applications that expect an IP address to be 32 bits and stored in an unsigned integer, you can copy the first four bytes of the *ipNumber* array using the **CopyMemory** function or equivalent. Note that if this is done, your application should always check the *ipFamily* member first to make sure that it is actually an IPv4 address.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Domain Name Service Constants

Value	Constant	Description
0	DNS_RECORD_NONE	No record type
1	DNS_RECORD_ADDRESS	Host address
2	DNS_RECORD_NS	Authoritative nameserver
5	DNS_RECORD_CNAME	Canonical name (alias)
6	DNS_RECORD_SOA	Start of Authority
11	DNS_RECORD_WKS	Well known services
12	DNS_RECORD_PTR	Domain name
13	DNS_RECORD_HINFO	Host information
14	DNS_RECORD_MINFO	Mailbox information
15	DNS_RECORD_MX	Mail exchange host
16	DNS_RECORD_TXT	Text strings
29	DNS_RECORD_LOC	Location information
100	DNS_RECORD_UINFO	User information
101	DNS_RECORD_UID	User ID
102	DNS_RECORD_GID	Group ID

SocketTools Library Error Codes

Value	Constant	Description
0x80042711	ST_ERROR_NOT_HANDLE_OWNER	Handle not owned by the current thread
0x80042712	ST_ERROR_FILE_NOT_FOUND	The specified file or directory does not exist
0x80042713	ST_ERROR_FILE_NOT_CREATED	The specified file could not be created
0x80042714	ST_ERROR_OPERATION_CANCELED	The blocking operation has been canceled
0x80042715	ST_ERROR_INVALID_FILE_TYPE	The specified file is a block or character device, not a regular file
0x80042716	ST_ERROR_INVALID_DEVICE	The specified device or address does not exist
0x80042717	ST_ERROR_TOO_MANY_PARAMETERS	The maximum number of function parameters has been exceeded
0x80042718	ST_ERROR_INVALID_FILE_NAME	The specified file name contains invalid characters or is too long
0x80042719	ST_ERROR_INVALID_FILE_HANDLE	Invalid file handle passed to function
0x8004271A	ST_ERROR_FILE_READ_FAILED	Unable to read data from the specified file
0x8004271B	ST_ERROR_FILE_WRITE_FAILED	Unable to write data to the specified file
0x8004271C	ST_ERROR_OUT_OF_MEMORY	Out of memory
0x8004271D	ST_ERROR_ACCESS_DENIED	Access denied
0x8004271E	ST_ERROR_INVALID_PARAMETER	Invalid argument passed to function
0x8004271F	ST_ERROR_CLIPBOARD_UNAVAILABLE	The system clipboard is currently unavailable
0x80042720	ST_ERROR_CLIPBOARD_EMPTY	The system clipboard is empty or does not contain any text data
0x80042721	ST_ERROR_FILE_EMPTY	The specified file does not contain any data
0x80042722	ST_ERROR_FILE_EXISTS	The specified file already exists
0x80042723	ST_ERROR_END_OF_FILE	End of file
0x80042724	ST_ERROR_DEVICE_NOT_FOUND	The specified device could not be found
0x80042725	ST_ERROR_DIRECTORY_NOT_FOUND	The specified directory could not be found
0x80042726	ST_ERROR_INVALID_BUFFER	Invalid memory address passed to

		function
0x80042728	ST_ERROR_NO_HANDLES	No more handles available to this process
0x80042733	ST_ERROR_OPERATION_WOULD_BLOCK	The specified operation would block the current thread
0x80042734	ST_ERROR_OPERATION_IN_PROGRESS	A blocking operation is currently in progress
0x80042735	ST_ERROR_ALREADY_IN_PROGRESS	The specified operation is already in progress
0x80042736	ST_ERROR_INVALID_HANDLE	Invalid handle passed to function
0x80042737	ST_ERROR_INVALID_ADDRESS	Invalid network address specified
0x80042738	ST_ERROR_INVALID_SIZE	Datagram is too large to fit in specified buffer
0x80042739	ST_ERROR_INVALID_PROTOCOL	Invalid network protocol specified
0x8004273A	ST_ERROR_PROTOCOL_NOT_AVAILABLE	The specified network protocol is not available
0x8004273B	ST_ERROR_PROTOCOL_NOT_SUPPORTED	The specified protocol is not supported
0x8004273C	ST_ERROR_SOCKET_NOT_SUPPORTED	The specified socket type is not supported
0x8004273D	ST_ERROR_INVALID_OPTION	The specified option is invalid
0x8004273E	ST_ERROR_PROTOCOL_FAMILY	Specified protocol family is not supported
0x8004273F	ST_ERROR_PROTOCOL_ADDRESS	The specified address is invalid for this protocol family
0x80042740	ST_ERROR_ADDRESS_IN_USE	The specified address is in use by another process
0x80042741	ST_ERROR_ADDRESS_UNAVAILABLE	The specified address cannot be assigned
0x80042742	ST_ERROR_NETWORK_UNAVAILABLE	The networking subsystem is unavailable
0x80042743	ST_ERROR_NETWORK_UNREACHABLE	The specified network is unreachable
0x80042744	ST_ERROR_NETWORK_RESET	Network dropped connection on remote reset
0x80042745	ST_ERROR_CONNECTION_ABORTED	Connection was aborted due to timeout or other failure
0x80042746	ST_ERROR_CONNECTION_RESET	Connection was reset by remote network
0x80042747	ST_ERROR_OUT_OF_BUFFERS	No buffer space is available
0x80042748	ST_ERROR_ALREADY_CONNECTED	Connection already established with

		remote host
0x80042749	ST_ERROR_NOT_CONNECTED	No connection established with remote host
0x8004274A	ST_ERROR_CONNECTION_SHUTDOWN	Unable to send or receive data after connection shutdown
0x8004274C	ST_ERROR_OPERATION_TIMEOUT	The specified operation has timed out
0x8004274D	ST_ERROR_CONNECTION_REFUSED	The connection has been refused by the remote host
0x80042750	ST_ERROR_HOST_UNAVAILABLE	The specified host is unavailable
0x80042751	ST_ERROR_HOST_UNREACHABLE	Remote host is unreachable
0x80042753	ST_ERROR_TOO_MANY_PROCESSES	Too many processes are using the networking subsystem
0x80042755	ST_ERROR_TOO_MANY_THREADS	Too many threads have been created by the current process
0x80042756	ST_ERROR_TOO_MANY_SESSIONS	Too many client sessions have been created by the current process
0x80042762	ST_ERROR_INTERNAL_FAILURE	An unexpected internal error has occurred
0x8004276B	ST_ERROR_NETWORK_NOT_READY	Network subsystem is not ready for communication
0x8004276C	ST_ERROR_INVALID_VERSION	This version of the operating system is not supported
0x8004276D	ST_ERROR_NETWORK_NOT_INITIALIZED	The networking subsystem has not been initialized
0x80042775	ST_ERROR_REMOTE_SHUTDOWN	The remote host has initiated a graceful shutdown sequence
0x80042AF9	ST_ERROR_INVALID_HOSTNAME	The specified hostname is invalid or could not be resolved
0x80042AFA	ST_ERROR_HOSTNAME_NOT_FOUND	The specified hostname could not be found
0x80042AFB	ST_ERROR_HOSTNAME_REFUSED	Unable to resolve hostname, request refused
0x80042AFC	ST_ERROR_HOSTNAME_NOT_RESOLVED	Unable to resolve hostname, no address for specified host
0x80042EE1	ST_ERROR_INVALID_LICENSE	The license for this product is invalid
0x80042EE2	ST_ERROR_PRODUCT_NOT_LICENSED	This product is not licensed to perform this operation
0x80042EE3	ST_ERROR_NOT_IMPLEMENTED	This function has not been implemented on this platform
0x80042EE4	ST_ERROR_UNKNOWN_LOCALHOST	Unable to determine local host name

0x80042EE5	ST_ERROR_INVALID_HOSTADDRESS	Invalid host address specified
0x80042EE6	ST_ERROR_INVALID_SERVICE_PORT	Invalid service port number specified
0x80042EE7	ST_ERROR_INVALID_SERVICE_NAME	Invalid or unknown service name specified
0x80042EE8	ST_ERROR_INVALID_EVENTID	Invalid event identifier specified
0x80042EE9	ST_ERROR_OPERATION_NOT_BLOCKING	No blocking operation in progress on this socket
0x80042F45	ST_ERROR_SECURITY_NOT_INITIALIZED	Unable to initialize security interface for this process
0x80042F46	ST_ERROR_SECURITY_CONTEXT	Unable to establish security context for this session
0x80042F47	ST_ERROR_SECURITY_CREDENTIALS	Unable to open client certificate store or establish client credentials
0x80042F48	ST_ERROR_SECURITY_CERTIFICATE	Unable to validate the certificate chain for this session
0x80042F49	ST_ERROR_SECURITY_DECRYPTION	Unable to decrypt data stream
0x80042F4A	ST_ERROR_SECURITY_ENCRYPTION	Unable to encrypt data stream
0x80042FA9	ST_ERROR_OPERATION_NOT_SUPPORTED	The specified operation is not supported
0x80042FAA	ST_ERROR_INVALID_PROTOCOL_VERSION	Invalid application protocol version specified
0x80042FAB	ST_ERROR_NO_SERVER_RESPONSE	No data returned from server
0x80042FAC	ST_ERROR_INVALID_SERVER_RESPONSE	Invalid data returned from server
0x80042FAD	ST_ERROR_UNEXPECTED_SERVER_RESPONSE	Unexpected response code returned from server
0x80042FAE	ST_ERROR_SERVER_TRANSACTION_FAILED	Server transaction failed
0x80042FAF	ST_ERROR_SERVICE_UNAVAILABLE	The service is currently unavailable
0x80042FB0	ST_ERROR_SERVICE_NOT_READY	The service is not ready, try again later
0x80042FB1	ST_ERROR_SERVER_RESYNC_FAILED	Unable to resynchronize with server
0x80042FB2	ST_ERROR_INVALID_PROXY_TYPE	Invalid proxy server type specified
0x80042FB3	ST_ERROR_PROXY_REQUIRED	Resource must be accessed through specified proxy
0x80042FB4	ST_ERROR_INVALID_PROXY_LOGIN	Unable to login to proxy server using specified credentials
0x80042FB5	ST_ERROR_PROXY_RESYNC_FAILED	Unable to resynchronize with proxy server
0x80042FB6	ST_ERROR_INVALID_COMMAND	Invalid command specified
0x80042FB7	ST_ERROR_INVALID_COMMAND_PARAMETER	Invalid command parameter specified

0x80042FB8	ST_ERROR_INVALID_COMMAND_SEQUENCE	Invalid command sequence specified
0x80042FB9	ST_ERROR_COMMAND_NOT_IMPLEMENTED	Specified command not implemented on this server
0x80042FBA	ST_ERROR_COMMAND_NOT_AUTHORIZED	Specified command not authorized for the current user
0x80042FBB	ST_ERROR_COMMAND_ABORTED	Specified command was aborted by the remote host
0x80042FBC	ST_ERROR_OPTION_NOT_SUPPORTED	The specified option is not supported on this server
0x80042FBD	ST_ERROR_REQUEST_NOT_COMPLETED	The current client request has not been completed
0x80042FBE	ST_ERROR_INVALID_USERNAME	The specified username is invalid
0x80042FBF	ST_ERROR_INVALID_PASSWORD	The specified password is invalid
0x80042FC0	ST_ERROR_INVALID_ACCOUNT	The specified account name is invalid
0x80042FC1	ST_ERROR_ACCOUNT_REQUIRED	Account name has not been specified
0x80042FC2	ST_ERROR_INVALID_AUTHENTICATION_TYPE	Invalid authentication protocol specified
0x80042FC3	ST_ERROR_AUTHENTICATION_REQUIRED	User authentication is required
0x80042FC4	ST_ERROR_PROXY_AUTHENTICATION_REQUIRED	Proxy authentication required
0x80042FC5	ST_ERROR_ALREADY_AUTHENTICATED	User has already been authenticated
0x80042FC6	ST_ERROR_AUTHENTICATION_FAILED	Unable to authenticate the specified user
0x80042FDB	ST_ERROR_NETWORK_ADAPTER	Unable to determine network adapter configuration
0x80042FDC	ST_ERROR_INVALID_RECORD_TYPE	Invalid record type specified
0x80042FDD	ST_ERROR_INVALID_RECORD_NAME	Invalid record name specified
0x80042FDE	ST_ERROR_INVALID_RECORD_DATA	Invalid record data specified
0x80042FDF	ST_ERROR_CONNECTION_OPEN	Data connection already established
0x80042FE0	ST_ERROR_CONNECTION_CLOSED	Server closed data connection
0x80042FE1	ST_ERROR_CONNECTION_PASSIVE	Data connection is passive
0x80042FE2	ST_ERROR_CONNECTION_FAILED	Unable to open data connection to server
0x80042FE3	ST_ERROR_INVALID_SECURITY_LEVEL	Data connection cannot be opened with this security setting
0x80042FE4	ST_ERROR_CACHED_TLS_REQUIRED	Data connection requires cached TLS session
0x80042FE5	ST_ERROR_DATA_READ_ONLY	Data connection is read-only
0x80042FE6	ST_ERROR_DATA_WRITE_ONLY	Data connection is write-only

0x80042FE7	ST_ERROR_END_OF_DATA	End of data
0x80042FE8	ST_ERROR_REMOTE_FILE_UNAVAILABLE	Remote file is unavailable
0x80042FE9	ST_ERROR_INSUFFICIENT_STORAGE	Insufficient storage on server
0x80042FEA	ST_ERROR_STORAGE_ALLOCATION	File exceeded storage allocation on server
0x80042FEB	ST_ERROR_DIRECTORY_EXISTS	The specified directory already exists
0x80042FEC	ST_ERROR_DIRECTORY_EMPTY	No files returned by the server for the specified directory
0x80042FED	ST_ERROR_END_OF_DIRECTORY	End of directory listing
0x80042FEE	ST_ERROR_UNKNOWN_DIRECTORY_FORMAT	Unknown directory format
0x80042FEF	ST_ERROR_INVALID_RESOURCE	Invalid resource name specified
0x80042FF0	ST_ERROR_RESOURCE_REDIRECTED	The specified resource has been redirected
0x80042FF1	ST_ERROR_RESOURCE_RESTRICTED	Access to this resource has been restricted
0x80042FF2	ST_ERROR_RESOURCE_NOT_MODIFIED	The specified resource has not been modified
0x80042FF3	ST_ERROR_RESOURCE_NOT_FOUND	The specified resource cannot be found
0x80042FF4	ST_ERROR_RESOURCE_CONFLICT	Request could not be completed due to the current state of the resource
0x80042FF5	ST_ERROR_RESOURCE_REMOVED	The specified resource has been permanently removed from this server
0x80042FF6	ST_ERROR_CONTENT_LENGTH_REQUIRED	Request must include the content length
0x80042FF7	ST_ERROR_REQUEST_PRECONDITION	Request could not be completed due to server precondition
0x80042FF8	ST_ERROR_UNSUPPORTED_MEDIA_TYPE	Request specified an unsupported media type
0x80042FF9	ST_ERROR_INVALID_CONTENT_RANGE	Content range specified for this resource is invalid
0x80042FFA	ST_ERROR_INVALID_MESSAGE_PART	Message is not multipart or an invalid message part was specified
0x80042FFB	ST_ERROR_INVALID_MESSAGE_HEADER	The specified message header is invalid or has not been defined
0x80042FFC	ST_ERROR_INVALID_MESSAGE_BOUNDARY	The multipart message boundary has not been defined
0x80042FFD	ST_ERROR_NO_FILE_ATTACHMENT	The current message part does not contain a file attachment

0x80042FFE	ST_ERROR_UNKNOWN_FILE_TYPE	The specified file type could not be determined
0x80042FFF	ST_ERROR_DATA_NOT_ENCODED	The specified data block could not be encoded
0x80043000	ST_ERROR_DATA_NOT_DECODED	The specified data block could not be decoded
0x80043001	ST_ERROR_FILE_NOT_ENCODED	The specified file could not be encoded
0x80043002	ST_ERROR_FILE_NOT_DECODED	The specified file could not be decoded
0x80043003	ST_ERROR_NO_MESSAGE_TEXT	No message text
0x80043004	ST_ERROR_INVALID_CHARACTER_SET	Invalid character set specified
0x80043005	ST_ERROR_INVALID_ENCODING_TYPE	Invalid encoding type specified
0x80043006	ST_ERROR_INVALID_MESSAGE_NUMBER	Invalid message number specified
0x80043007	ST_ERROR_NO_RETURN_ADDRESS	No valid return address specified
0x80043008	ST_ERROR_NO_VALID_RECIPIENTS	No valid recipients specified
0x80043009	ST_ERROR_INVALID_RECIPIENT	The specified recipient address is invalid
0x8004300A	ST_ERROR_RELAY_NOT_AUTHORIZED	The specified domain is invalid or server will not relay messages
0x8004300B	ST_ERROR_MAILBOX_UNAVAILABLE	Specified mailbox is currently unavailable
0x8004300C	ST_ERROR_MAILBOX_READONLY	The selected mailbox cannot be modified
0x8004300D	ST_ERROR_MAILBOX_NOT_SELECTED	No mailbox has been selected
0x8004300E	ST_ERROR_INVALID_MAILBOX	Specified mailbox is invalid
0x8004300F	ST_ERROR_INVALID_DOMAIN	The specified domain name is invalid or not recognized
0x80043010	ST_ERROR_INVALID_SENDER	The specified sender address is invalid or not recognized
0x80043011	ST_ERROR_MESSAGE_NOT_DELIVERED	Message not delivered to any of the specified recipients
0x80043012	ST_ERROR_END_OF_MESSAGE_DATA	No more message data available to be read
0x80043013	ST_ERROR_INVALID_MESSAGE_SIZE	The specified message size is invalid
0x80043014	ST_ERROR_MESSAGE_NOT_CREATED	The message could not be created in the specified mailbox
0x80043015	ST_ERROR_NO_MORE_MAILBOXES	No more mailboxes exist on this server

0x80043016	ST_ERROR_INVALID_EMULATION_TYPE	The specified terminal emulation type is invalid
0x80043017	ST_ERROR_INVALID_FONT_HANDLE	The specified font handle is invalid
0x80043018	ST_ERROR_INVALID_FONT_NAME	The specified font name is invalid or unavailable
0x80043019	ST_ERROR_INVALID_PACKET_SIZE	The specified packet size is invalid
0x8004301A	ST_ERROR_INVALID_PACKET_DATA	The specified packet data is invalid
0x8004301B	ST_ERROR_INVALID_PACKET_ID	The unique packet identifier is invalid
0x8004301C	ST_ERROR_PACKET_TTL_EXPIRED	The specified packet time-to-live period has expired
0x8004301D	ST_ERROR_INVALID_NEWSGROUP	Invalid newsgroup specified
0x8004301E	ST_ERROR_NO_NEWSGROUP_SELECTED	No newsgroup selected
0x8004301F	ST_ERROR_EMPTY_NEWSGROUP	No articles in specified newsgroup
0x80043020	ST_ERROR_INVALID_ARTICLE	Invalid article number specified
0x80043021	ST_ERROR_NO_ARTICLE_SELECTED	No article selected in the current newsgroup
0x80043022	ST_ERROR_FIRST_ARTICLE	First article in current newsgroup
0x80043023	ST_ERROR_LAST_ARTICLE	Last article in current newsgroup
0x80043024	ST_ERROR_ARTICLE_EXISTS	Unable to transfer article, article already exists
0x80043025	ST_ERROR_ARTICLE_REJECTED	Unable to transfer article, article rejected
0x80043026	ST_ERROR_ARTICLE_TRANSFER_FAILED	Article transfer failed
0x80043027	ST_ERROR_ARTICLE_POSTING_DENIED	Posting is not permitted on this server
0x80043028	ST_ERROR_ARTICLE_POSTING_FAILED	Posting is not permitted on this server
0x80043029	ST_ERROR_INVALID_DATE_FORMAT	The specified date format is not recognized
0x8004302A	ST_ERROR_FEATURE_NOT_SUPPORTED	The specified feature is not supported on this server
0x8004302B	ST_ERROR_INVALID_FORM_HANDLE	The specified form handle is invalid or a form has not been created
0x8004302C	ST_ERROR_INVALID_FORM_ACTION	The specified form action is invalid or has not been specified
0x8004302D	ST_ERROR_INVALID_FORM_METHOD	The specified form method is invalid or not supported
0x8004302E	ST_ERROR_INVALID_FORM_TYPE	The specified form type is invalid or not supported
0x8004302F	ST_ERROR_INVALID_FORM_FIELD	The specified form field name is invalid or does not exist

0x80043030	ST_ERROR_EMPTY_FORM	The specified form does not contain any field values
0x80043031	ST_ERROR_MAXIMUM_CONNECTIONS	The maximum number of client connections exceeded
0x80043032	ST_ERROR_THREAD_CREATION_FAILED	Unable to create a new thread for the current process
0x80043033	ST_ERROR_INVALID_THREAD_HANDLE	The specified thread handle is no longer valid
0x80043034	ST_ERROR_THREAD_TERMINATED	The specified thread has been terminated
0x80043035	ST_ERROR_THREAD_DEADLOCK	The operation would result in the current thread becoming deadlocked
0x80043036	ST_ERROR_INVALID_CLIENT_MONIKER	The specified moniker is not associated with any client session
0x80043037	ST_ERROR_CLIENT_MONIKER_EXISTS	The specified moniker has been assigned to another client session
0x80043038	ST_ERROR_SERVER_INACTIVE	The specified server is not listening for client connections
0x80043039	ST_ERROR_SERVER_SUSPENDED	The specified server is suspended and not accepting client connections
0x8004303A	ST_ERROR_NO_MESSAGE_STORE	No message store has been specified
0x8004303B	ST_ERROR_MESSAGE_STORE_CHANGED	The message store has changed since it was last accessed
0x8004303C	ST_ERROR_MESSAGE_NOT_FOUND	No message was found that matches the specified criteria
0x8004303D	ST_ERROR_MESSAGE_DELETED	The specified message has been deleted
0x8004303E	ST_ERROR_FILE_CHECKSUM_MISMATCH	The local and remote file checksums do not match
0x8004303F	ST_ERROR_FILE_SIZE_MISMATCH	The local and remote file sizes do not match
0x80043040	ST_ERROR_INVALID_FEED_URL	The news feed URL is invalid or specifies an unsupported protocol
0x80043041	ST_ERROR_INVALID_FEED_FORMAT	The internal format of the news feed is invalid
0x80043042	ST_ERROR_INVALID_FEED_VERSION	This version of the news feed is not supported
0x80043043	ST_ERROR_CHANNEL_EMPTY	There are no valid items found in this news feed
0x80043044	ST_ERROR_INVALID_ITEM_NUMBER	The specified channel item identifier is invalid

0x80043045	ST_ERROR_ITEM_NOT_FOUND	The specified channel item could not be found
0x80043046	ST_ERROR_ITEM_EMPTY	The specified channel item does not contain any data
0x80043047	ST_ERROR_INVALID_ITEM_PROPERTY	The specified item property name is invalid
0x80043048	ST_ERROR_ITEM_PROPERTY_NOT_FOUND	The specified item property has not been defined
0x80043049	ST_ERROR_INVALID_CHANNEL_TITLE	The channel title is invalid or has not been defined
0x8004304A	ST_ERROR_INVALID_CHANNEL_LINK	The channel hyperlink is invalid or has not been defined
0x8004304B	ST_ERROR_INVALID_CHANNEL_DESCRIPTION	The channel description is invalid or has not been defined
0x8004304C	ST_ERROR_INVALID_ITEM_TEXT	The description for an item is invalid or has not been defined
0x8004304D	ST_ERROR_INVALID_ITEM_LINK	The hyperlink for an item is invalid or has not been defined
0x8004304E	ST_ERROR_INVALID_SERVICE_TYPE	The specified service type is invalid
0x8004304F	ST_ERROR_SERVICE_SUSPENDED	Access to the specified service has been suspended
0x80043050	ST_ERROR_SERVICE_RESTRICTED	Access to the specified service has been restricted
0x80043051	ST_ERROR_INVALID_PROVIDER_NAME	The specified provider name is invalid or unknown
0x80043052	ST_ERROR_INVALID_PHONE_NUMBER	The specified phone number is invalid or not supported in this region
0x80043053	ST_ERROR_GATEWAY_NOT_FOUND	A message gateway cannot be found for the specified provider
0x80043054	ST_ERROR_MESSAGE_TOO_LONG	The message exceeds the maximum number of characters permitted
0x80043055	ST_ERROR_INVALID_PROVIDER_DATA	The request returned invalid or incomplete service provider data
0x80043056	ST_ERROR_INVALID_GATEWAY_DATA	The request returned invalid or incomplete message gateway data
0x80043057	ST_ERROR_MULTIPLE_PROVIDERS	The request has returned multiple service providers
0x80043058	ST_ERROR_PROVIDER_NOT_FOUND	The specified service provider could not be found
0x80043059	ST_ERROR_INVALID_MESSAGE_SERVICE	The specified message is not supported with this service type

0x8004305A	ST_ERROR_INVALID_MESSAGE_FORMAT	The specified message format is invalid
0x8004305B	ST_ERROR_INVALID_CONFIGURATION	The specified configuration options are invalid
0x8004305C	ST_ERROR_SERVER_ACTIVE	The requested action is not permitted while the server is active
0x8004305D	ST_ERROR_SERVER_PORT_BOUND	Unable to obtain exclusive use of the specified local port
0x8004305E	ST_ERROR_INVALID_CLIENT_SESSION	The specified client identifier is invalid for this session
0x8004305F	ST_ERROR_CLIENT_NOT_IDENTIFIED	The specified client has not provided user credentials
0x80043060	ST_ERROR_INVALID_CLIENT_STATE	The requested action cannot be performed at this time
0x80043061	ST_ERROR_INVALID_RESULT_CODE	The specified result code is not valid for this protocol
0x80043062	ST_ERROR_COMMAND_REQUIRED	The specified command is required and cannot be disabled
0x80043063	ST_ERROR_COMMAND_DISABLED	The specified command has been disabled
0x80043064	ST_ERROR_COMMAND_SEQUENCE	The command cannot be processed at this time
0x80043065	ST_ERROR_COMMAND_COMPLETED	The previous command has completed
0x80043066	ST_ERROR_INVALID_PROGRAM_NAME	The specified program name is invalid or unrecognized
0x80043067	ST_ERROR_INVALID_REQUEST_HEADER	The request header contains one or more invalid values
0x80043068	ST_ERROR_INVALID_VIRTUAL_HOST	The specified virtual host name is invalid
0x80043069	ST_ERROR_VIRTUAL_HOST_NOT_FOUND	The specified virtual host does not exist
0x8004306A	ST_ERROR_TOO_MANY_VIRTUAL_HOSTS	Too many virtual hosts created for this server
0x8004306B	ST_ERROR_INVALID_VIRTUAL_PATH	The specified virtual path name is invalid
0x8004306C	ST_ERROR_VIRTUAL_PATH_NOT_FOUND	The specified virtual path does not exist
0x8004306D	ST_ERROR_TOO_MANY_VIRTUAL_PATHS	Too many virtual paths created for this server
0x8004306E	ST_ERROR_INVALID_TASK	The asynchronous task identifier is

		invalid
0x8004306F	ST_ERROR_TASK_ACTIVE	The asynchronous task has not finished
0x80043070	ST_ERROR_TASK_QUEUED	The asynchronous task has been queued
0x80043071	ST_ERROR_TASK_SUSPENDED	The asynchronous task has been suspended
0x80043072	ST_ERROR_TASK_FINISHED	The asynchronous task has finished
0x80043073	ST_ERROR_INVALID_ACCOUNT_UUID	The account unique identifier is invalid
0x80043074	ST_ERROR_INVALID_ACCOUNT_ID	The application account identifier is invalid
0x80043075	ST_ERROR_INVALID_PRODUCT_ID	The product identifier identifier is invalid
0x80043076	ST_ERROR_INVALID_SERIAL_NUMBER	The product serial number is invalid
0x80043077	ST_ERROR_INVALID_APPID	The application identifier is invalid
0x80043078	ST_ERROR_INVALID_APIKEY	The application key is invalid
0x80043079	ST_ERROR_ACCOUNT_EXISTS	The application account identifier already exists
0x8004307A	ST_ERROR_ACCOUNT_NOT_CREATED	The application account identifier was not created
0x8004307B	ST_ERROR_ACCOUNT_NOT_FOUND	The application account identifier was not found
0x8004307C	ST_ERROR_ACCOUNT_NOT_EXPIRED	Access to this account has not expired
0x8004307D	ST_ERROR_ACCOUNT_NOT_UPDATED	The application account could not be updated
0x8004307E	ST_ERROR_ACCOUNT_EXPIRED	Access to this account has expired
0x8004307F	ST_ERROR_ACCOUNT_REVOKED	Access to this account has been revoked
0x80043080	ST_ERROR_APIKEY_NOT_CREATED	The application key could not be created
0x80043081	ST_ERROR_APIKEY_NOT_FOUND	The application key could not be found
0x80043082	ST_ERROR_APIKEY_NOT_EXPIRED	The application key has not expired
0x80043083	ST_ERROR_APIKEY_NOT_UNIQUE	The application key identifier is not unique
0x80043084	ST_ERROR_APIKEY_NOT_UPDATED	They application key could not be updated
0x80043085	ST_ERROR_APIKEY_NOT_DELETED	The application key could not be deleted

0x80043086	ST_ERROR_APIKEY_EXISTS	The application key already exists
0x80043087	ST_ERROR_APIKEY_EXPIRED	The application key has expired and must be refreshed
0x80043088	ST_ERROR_APIKEY_REVOKED	The application key has been revoked
0x80043089	ST_ERROR_APIKEY_APPID	The application was not found or was not specified
0x8004308A	ST_ERROR_INVALID_TOKEN	The access token is invalid or was not specified
0x8004308B	ST_ERROR_TOKEN_NOT_CREATED	The access token could not be created
0x8004308C	ST_ERROR_TOKEN_NOT_FOUND	The access token could not be found
0x8004308D	ST_ERROR_TOKEN_NOT_EXPIRED	The access token has not expired
0x8004308E	ST_ERROR_TOKEN_NOT_UPDATED	The access token was not updated
0x8004308F	ST_ERROR_TOKEN_NOT_DELETED	The access token could not be deleted
0x80043090	ST_ERROR_TOKEN_EXPIRED	The access token has expired and must be refreshed
0x80043091	ST_ERROR_TOKEN_REVOKED	The access token has been revoked
0x80043092	ST_ERROR_NO_APIKEYS_FOUND	No application keys found for this account
0x80043093	ST_ERROR_NO_TOKENS_FOUND	No access tokens found for this application key
0x80043094	ST_ERROR_NO_TOKENS_REVOKED	No access tokens have been revoked
0x80043095	ST_ERROR_INVALID_STORAGE_OBJECT	Invalid storage object identifier
0x80043096	ST_ERROR_STORAGE_OBJECT_READONLY	The storage object is read-only
0x80043097	ST_ERROR_STORAGE_OBJECT_EXPIRED	Access to the storage object has expired
0x80043098	ST_ERROR_STORAGE_OBJECT_SIZE	The storage object size exceeds storage limits
0x80043099	ST_ERROR_STORAGE_OBJECT_DIGEST	The storage object digest is invalid or cannot be computed
0x8004309A	ST_ERROR_STORAGE_OBJECT_EXISTS	A storage object with this label already exists
0x8004309B	ST_ERROR_STORAGE_OBJECT_MODIFIED	A storage object with this label has been modified
0x8004309C	ST_ERROR_STORAGE_OBJECT_NOT_OWNER	The current user is not the storage object owner
0x8004309D	ST_ERROR_STORAGE_OBJECT_NOT_FOUND	The specified storage object does not exist

0x8004309E	ST_ERROR_STORAGE_OBJECT_NOT_CREATED	The storage object was not created
0x8004309F	ST_ERROR_STORAGE_OBJECT_NOT_MODIFIED	The storage object was not modified
0x800430A0	ST_ERROR_STORAGE_OBJECT_NOT_RENAMED	The storage object was not renamed
0x800430A1	ST_ERROR_STORAGE_FOLDER_EMPTY	The storage folder does not contain any objects
0x800430A2	ST_ERROR_STORAGE_ACCOUNT_QUOTA	The storage account has exceeded its quota
0x800430A3	ST_ERROR_STORAGE_ACCOUNT_LIMIT	The storage account has exceeded its object limit
0x800430A4	ST_ERROR_INVALID_STORAGE_TYPE	The specified storage type is invalid
0x800430A5	ST_ERROR_INVALID_STORAGE_PROVIDER	The specified storage provider is not available
0x800430A6	ST_ERROR_INVALID_STORAGE_REGION	The specified storage region is not available
0x800430A7	ST_ERROR_INVALID_STORAGE_FOLDER	The storage folder does not exist or cannot be accessed
0x800430A8	ST_ERROR_INVALID_STORAGE_LABEL	The storage object label is invalid or undefined

File Transfer Protocol Class Library

Transfer files between a local and server and perform common file management methods on the server.

Reference

- [Class Methods](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

Class Name	CFtpClient
File Name	CSFTPV10.DLL
Version	10.0.1468.2518
LibID	C01B14F0-7091-429D-A690-B99D635CB8AB
Import Library	CSFTPV10.LIB
Dependencies	None
Standards	RFC 959, RFC 1579, RFC 2228

Overview

The File Transfer Protocol (FTP) library provides a comprehensive API which supports both high level operations, such as uploading or downloading files, as well as a collection of lower-level file I/O functions. In addition to file transfers, an application can create, rename and delete files and directories, search for files using wildcards and perform other common file management functions.

Files can be stored on the local file system or in memory, depending on the needs of your application and multiple file transfers be performed using a single function call. The library can also be used to manage files on the server and supports many of the common protocol extensions that can be used to access the remote file system. It understands a number of different directory listing formats, including those typically used with UNIX and Linux based systems, Windows server platforms, NetWare servers and VMS systems.

This library supports active and passive mode file transfers, firewall compatibility options, proxy servers and secure file transfers using the standard SSL/TLS and SFTP protocols. Secure file transfers support implicit and explicit SSL sessions, client certificates and up to 256-bit AES encryption.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This class is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical

updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

File Transfer Protocol Class Methods

Class	Description
CFtpClient	Constructor which initializes the current instance of the class
~CFtpClient	Destructor which releases resources allocated by the class
Method	Description
Allocate	Allocate the specified number of bytes on the server
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified client handle to another thread
Cancel	Cancel the current blocking operation
ChangeDirectory	Change the current working directory on the server
ChangeDirectoryUp	Change the current working directory on the server
CloseDirectory	Close the open directory on the server
CloseFile	Close the current file on the server
Command	Send a command to the server
Connect	Establish a client connection with a server
ConnectUrl	Establish a client connection using the specified URL
CreateDirectory	Create the specified directory on the server
CreateSecurityCredentials	Create a new security credentials structure
DeleteFile	Delete a file from the server
DeleteSecurityCredentials	Delete a previously created security credentials structure
DetachHandle	Detach the handle for the current instance of this class
DisableEvents	Disable event notification
DisableTrace	Disable logging of socket function calls to the trace log
Disconnect	Disconnect the client session from the server
DownloadFile	Download a file from the server to the local system
EnableEvents	Enable event notification
EnableFeature	Enable the specified feature in the client
EnableTrace	Enable logging of socket function calls to a file
FreezeEvents	Suspend and resume event handling by the client
FtpEventProc	Callback method that processes events generated by the client
GetActivePorts	Return the range of local port numbers used for active transfers
GetBufferSize	Return the size of an internal buffer used during data transfers
GetChannelMode	Return the mode for the specified communication channel
GetClientQuota	Return quota information for the current client session

GetData	Copy the contents of a remote file to a local buffer
GetDirectory	Get the current working directory on the server
GetDirectoryFormat	Get the format which is used by the server to list files
GetErrorString	Return a description for the specified error code
GetFeatures	Return the features available to the client
GetFile	Copy a file from the server to the local system
GetFileList	Return an unparsed list of files in a string buffer
GetFileNameEncoding	Return the character encoding used when sending a file name to the server
GetFilePermissions	Return the access permissions for the specified file
GetFileSize	Return the size of a file on the server
GetFileStatus	Return file status information from the server
GetFileTime	Return the modification time for the specified file on the server
GetFileType	Return the default file type for the current session
GetFirstFile	Return the first file from the file list returned by the server
GetHandle	Return the client handle used by this instance of the class
GetLastError	Return the last error code
GetMultipleFiles	Copy multiple files from the server to the local system
GetNextFile	Return the next file from the file list returned by the server
GetPriority	Return the current priority for file transfers
GetProxyType	Return the proxy type selected by the client
GetResultCode	Return the result code from the previous command
GetResultString	Return the result string from the previous command
GetSecurityInformation	Return security information about the current client connection
GetServerInformation	Get system information about the server
GetServerStatus	Return system status of server
GetServerTimeZone	Return the timezone offset in seconds for the current server
GetServerType	Return the type of operating system the server is running on
GetStatus	Return the current client status
GetText	Download the contents of a text file to a string buffer
GetTimeout	Return the number of seconds until an operation times out
GetTransferStatus	Return file transfer statistics
IsBlocking	Determine if the current operation is blocked
IsConnected	Determine if the client is connected to the server
IsInitialized	Determine if the class has been successfully initialized

IsReadable	Determine if the client can read data from the data channel
IsWritable	Determine if the client can write data to the data channel
Login	Login to the server
Logout	Logout from the server
MountStructure	Mount a structure (filesystem) on the server
OpenDirectory	Open the specified directory for reading
OpenFile	Open the specified file for reading or writing
ProxyConnect	Establish a connection with a proxy server
PutData	Create a file on the server using the contents of a local buffer
PutFile	Copy a file from the local system to the server
PutMultipleFiles	Copy multiple files from the local system to the server
PutText	Create a text file on the server from the contents of a string buffer
Read	Read data from the server
RegisterEvent	Register an event handler for the specified event
RegisterFileType	Associate a file name extension with a specific file type
RemoveDirectory	Remove a directory from the server
RenameFile	Rename a file on the server
Reset	Reset the client connection
SetActivePorts	Set the range of local port numbers used for active transfers
SetBufferSize	Set the size of an internal buffer used during data transfers
SetChannelMode	Change the security mode for the specified channel
SetDirectoryFormat	Set the format which is used by the server to list files
SetFeatures	Set the features which can be used by the client
SetFileMode	Set the current file mode
SetFileNameEncoding	Set the character encoding type used when sending a file name to the server
SetFilePermissions	Set the access permissions for the specified file
SetFileStructure	Set the current file data structure
SetFileTime	Set the modification time for the specified file on the server
SetFileType	Set the default file type for the current session
SetLastError	Set the last error code
SetPassiveMode	Set the server in passive mode
SetPriority	Set the priority for file transfers
SetTimeout	Set the number of seconds until an operation times out
ShowError	Display a message box with a description of the specified error
UploadFile	Upload a file from the local system to the server

ValidateUrl	Check the contents of a string to ensure it represents a valid URL
VerifyFile	Compare the contents of a local file against a file stored on the server
Write	Write data to the server

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CFtpClient::CFtpClient Method

CFtpClient();

The **CFtpClient** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **FtpInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[~CFtpClient](#), [IsInitialized](#)

CFtpClient::~CFtpClient

`~CFtpClient();`

The **CFtpClient** destructor releases resources allocated by the current instance of the **CFtpClient** object. It also uninitialized the library if there are no other concurrent uses of the class.

Remarks

When a **CFtpClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CFtpClient](#)

CFtpClient::Allocate Method

```
INT Allocate(  
    DWORD dwFileLength,  
    DWORD dwRecSize  
);
```

The **Allocate** method instructs the server to reserve sufficient storage to accommodate the new file being transferred.

Parameters

dwFileLength

The number of bytes to allocate storage for on the server.

dwRecSize

The maximum record or page size for the file. A value of zero indicates that the file does not have a record or page structure, and the parameter is ignored.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be called immediately before the **OpenFile** method.

This method is ignored by those servers which do not require that the maximum size of the file be declared beforehand. The most common FTP servers running under UNIX and Windows do not require that file space be pre-allocated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

See Also

[OpenFile](#), [SetFileMode](#), [SetFileStructure](#), [SetFileType](#)

CFtpClient::AttachHandle Method

```
VOID AttachHandle(  
    HCLIENT hClient  
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle already has been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the *hClient* parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[AttachThread](#), [DetachHandle](#), [GetHandle](#)

CFtpClient::AttachThread Method

```
DWORD AttachThread(  
    DWORD dwThreadId  
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

See Also

[AttachHandle](#), [Cancel](#), [Connect](#), [DetachHandle](#), [Disconnect](#), [GetHandle](#)

CFtpClient::Cancel Method

```
INT Cancel();
```

The **Cancel** method cancels any outstanding blocking operation in the client, causing the blocking method to fail. The application may then retry the operation or terminate the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[IsBlocking](#)

CFtpClient::ChangeDirectory Method

```
INT ChangeDirectory(  
    LPCTSTR lpszDirectory  
);
```

The **ChangeDirectory** method changes the current working directory for the client session.

Parameters

lpszDirectory

Points to a string that specifies the name of the directory. The file pathing and name conventions must be that of the server.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method uses the CWD command to change the current working directory. The user must have the appropriate permission to access the specified directory.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ChangeDirectoryUp](#), [CloseDirectory](#), [Command](#), [GetDirectory](#), [GetFirstFile](#), [GetNextFile](#), [GetResultCode](#), [GetResultString](#), [OpenDirectory](#)

CFtpClient::ChangeDirectoryUp Method

INT ChangeDirectoryUp();

The **ChangeDirectoryUp** method changes directory to the parent of the current working directory.

Parameters

None.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method sends the CDUP command to the server. This command is a special case of the CWD command, and is included to simplify transferring between directory trees on those operating systems which have different syntaxes for naming the parent directory. The current user must have the appropriate permission to access the specified directory.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ChangeDirectory](#), [Command](#), [GetDirectory](#), [GetResultCode](#), [GetResultString](#)

CFtpClient::CloseDirectory Method

```
INT CloseDirectory();
```

The **CloseDirectory** method closes the data socket connection to the server.

Parameters

None.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method must be called after all of the file information from the server has been returned. Because directory information is returned on the data channel, no file transfers can take place while a directory is being read.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[GetDirectoryFormat](#), [GetFileStatus](#), [GetFirstFile](#), [GetNextFile](#), [OpenDirectory](#), [SetDirectoryFormat](#)

CFtpClient::CloseFile Method

```
INT CloseFile();
```

The **CloseFile** method flushes the internal client buffers and closes the data socket connection to the server.

Parameters

None.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is `FTP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

If the file is opened for writing, all buffered data is written to the server before the socket is closed. This may cause the client to block until all of the data can be written. The client application should not perform any other action until the method returns.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

See Also

[OpenFile](#), [Read](#), [Write](#)

CFtpClient::Command Method

```
INT Command(  
    LPCTSTR LpszCommand,  
    LPCTSTR LpszParameter  
);
```

The **Command** method sends a command to the server, and returns the result code back to the caller. This method is typically used for site-specific commands not directly supported by the API.

Parameters

LpszCommand

The command which will be executed by the server.

LpszParameter

An optional command parameter. If the command requires more than one parameter, then they should be combined into a single string, with a space separating each parameter. If the command does not accept any parameters, this value may be NULL.

Return Value

If the method succeeds, the return value is the result code returned by the server. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should only be used when the application needs to send a custom, site-specific command or send a command that is not directly supported by this class. This method should never be used to issue a command that opens a data channel. If the application needs to transform data as it is being sent or received, and cannot use the **GetFile** or **PutFile** methods, then use the **OpenFile** method to open a data channel with the server.

By default, file names which are sent to the server using the **Command** method are sent as ANSI characters. If the Unicode version of the function is used, the file name will be converted from Unicode to ANSI using the current codepage. If the server supports UTF-8 encoded file names, the **SetFileNameEncoding** function can be used to specify that file names with non-ASCII characters should be sent as UTF-8 encoded values. It is important to note that this option is only available if the server advertises support for UTF-8 and permits that encoding type.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFile](#), [GetFileNameEncoding](#), [GetResultCode](#), [GetResultString](#), [OpenFile](#), [PutFile](#), [SetFileNameEncoding](#)

CFtpClient::Connect Method

```
BOOL Connect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    UINT nTimeout,  
    DWORD dwOptions,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **Connect** method establishes a connection with the specified server.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 21. For secure connections, the default port number is 990. If the secure port number is specified, an implicit SSL/TLS connection will be established by default.

lpszUserName

Points to a string that specifies the user name to be used to authenticate the current client session. If this parameter is NULL or an empty string, then the login is considered to be anonymous. Note that anonymous logins are not supported for secure connections using the SSH protocol.

lpszPassword

Points to a string that specifies the password to be used to authenticate the current client session. This parameter may be NULL or an empty string if no password is required for the specified user, or if no username has been specified.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
FTP_OPTION_PASSIVE	This option specifies the client should attempt to establish the data connection with the server. When the client uploads or downloads a file, normally the server establishes a second connection back to the client which is used to transfer the file data. However, if the local system is behind a firewall or a NAT router, the server may not be able to create

	<p>the data connection and the transfer will fail. By specifying this option, it forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.</p>
FTP_OPTION_FIREWALL	<p>This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.</p>
FTP_OPTION_NOAUTH	<p>This option specifies the server does not require authentication, or that it requires an alternate authentication method. When this option is used, the client connection is flagged as authenticated as soon as the connection to the server has been established. Note that using this option to bypass authentication may result in subsequent errors when attempting to retrieve a directory listing or transfer a file. It is recommended that you consult the technical reference documentation for the server to determine its specific authentication requirements.</p>
FTP_OPTION_KEEPAIVE	<p>This option specifies the client should attempt to keep the connection with the server active for an extended period of time. It is important to note that regardless of this option, the server may still choose to disconnect client sessions that are holding the command channel open but are not performing file transfers.</p>
FTP_OPTION_NOAUTHRSA	<p>This option specifies that RSA authentication should not be used with SSH-1 connections. This option is ignored with SSH-2 connections and should only be specified if required by the server. This option has no effect on standard or secure connections using SSL.</p>
FTP_OPTION_NOPWDNUL	<p>This option specifies the user password cannot be terminated with a null character. This option is ignored with SSH-2 connections and should only be specified if required by the server. This option has no effect on standard or secure connections using SSL.</p>
FTP_OPTION_NOREKEY	<p>This option specifies the client should never attempt a repeat key exchange with the server. Some SSH</p>

	<p>servers do not support rekeying the session, and this can cause the client to become non-responsive or abort the connection after being connected for an hour. This option has no effect on standard or secure connections using SSL.</p>
FTP_OPTION_COMPATSID	<p>This compatibility option changes how the session ID is handled during public key authentication with older SSH servers. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions. This option has no effect on standard or secure connections using SSL.</p>
FTP_OPTION_COMPATHMAC	<p>This compatibility option changes how the HMAC authentication codes are generated. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions. This option has no effect on standard or secure connections using SSL.</p>
FTP_OPTION_VIRTUALHOST	<p>This option specifies the server supports virtual hosting, where multiple domains are hosted by a server using the same external IP address. If this option is enabled, the client will send the HOST command to the server upon establishing a connection.</p>
FTP_OPTION_VERIFY	<p>This option specifies that file transfers should be automatically verified after the transfer has completed. If the server supports the XMD5 command, the transfer will be verified by calculating an MD5 hash of the file contents. If the server does not support the XMD5 command, but does support the XCRC command, the transfer will be verified by calculating a CRC32 checksum of the file contents. If neither the XMD5 or XCRC commands are supported, the transfer is verified by comparing the size of the file. Automatic file verification is only performed for binary mode transfers because of the end-of-line conversion that may occur when text files are uploaded or downloaded.</p>
FTP_OPTION_TRUSTEDSITE	<p>This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.</p>
FTP_OPTION_SECURE	<p>This option specifies the client should attempt to establish a secure connection with the server. This option is the same as specifying FTP_OPTION_SECURE_IMPLICIT which immediately performs the SSL/TLS protocol negotiation when</p>

	the connection is established.
FTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to immediately establish secure SSL/TLS connection with the server. This option is typically used when connecting to a server on port 990, which is the default port number used for FTPS.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should establish a standard connection to the server and then use the AUTH command to negotiate an explicit secure connection. This option is typically used when connecting to the server on ports other than 990.
FTP_OPTION_SECURE_SHELL	This option specifies the client should use the Secure Shell (SSH) protocol to establish the connection. This option will automatically be selected if the connection is established using port 22, the default port for SSH.
FTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
FTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established. This option also forces all connections to be outbound and enables the firewall compatibility features in the client.
FTP_OPTION_KEEPALIVE_DATA	This option specifies the client should attempt to keep the control connection active during a file transfer. Normally, when a data transfer is in progress, no additional commands are issued on the control channel until the transfer completes. Specifying this option automatically enables the FTP_OPTION_KEEPALIVE option and forces the client to continue to issue NOOP commands during the file transfer. This option only applies to FTP and FTPS connections and has no effect on connections using SFTP (SSH).
FTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname

	can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
FTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
FTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller files being uploaded or downloaded using fast network connections.
FTP_OPTION_TLS_REUSE	This option specifies that TLS session reuse should be enabled for secure connections. This option is only supported on Windows 8.1 or Windows Server 2012 R2 and later platforms, and it should only be used when explicitly required by the server. This option is not compatible with servers built using OpenSSL 1.0.2 and earlier versions which do not provide Extended Master Secret (EMS) support as outlined in RFC7627.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur. If this argument is NULL, then the client session will be blocking and no network events will be sent to the client.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the *hEventWnd* argument is NULL, this argument should be specified as WM_NULL.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The use of Windows event notification messages has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

To prevent this method from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **Connect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

If you specify an event notification window, then the client session will be asynchronous. When a message is posted to the notification window, the low word of the *lParam* parameter contains the

event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
FTP_EVENT_CONNECT	The control connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
FTP_EVENT_DISCONNECT	The server has closed the control connection to the client. The client should read any remaining data and disconnect.
FTP_EVENT_OPENFILE	The data connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
FTP_EVENT_CLOSEFILE	The server has closed the data connection to the client. The client should read any remaining data and close the data channel.
FTP_EVENT_READFILE	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_WRITEFILE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
FTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
FTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
FTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.
FTP_EVENT_GETFILE	This event is generated when a file download has completed. If multiple files are being downloaded, this event will be generated for each file.
FTP_EVENT_PUTFILE	This event is generated when a file upload has completed. If multiple files are being uploaded, this event will be generated for

each file.

To cancel asynchronous notification and return the client to a blocking mode, use the **DisableEvents** method.

If the `FTP_OPTION_KEEPALIVE` option is specified, a background worker thread will be created to monitor the command channel and periodically send NOOP commands to the server if no commands have been sent recently. This can prevent the server from terminating the client connection during idle periods where no commands are being issued. However, it is important to keep in mind that many servers can be configured to also limit the total amount of time a client can be connected to the server, as well as the amount of time permitted between file transfers. If the server does not respond to the NOOP command, this option will be automatically disabled for the remainder of the client session.

If the `FTP_OPTION_SECURE_EXPLICIT` option is specified, the client will establish a standard connection to the server and send the AUTH TLS command to the server. If the server does not accept this command, it will then send the AUTH SSL command. If both commands are rejected by the server, an explicit SSL session cannot be established. By default, both the command and data channels will be encrypted when a secure connection is established. To change this, use the **SetChannelMode** method.

The *dwOptions* argument can be used to specify the threading model that is used by the class when a connection is established. By default, the client session is initially attached to the thread that created it. From that point on, until the connection is terminated, only the owner may invoke methods in that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the `FTP_OPTION_FREETHREAD` option enables any thread to call methods in any instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the class and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a method in the class, it must ensure that no other thread will call another method at the same time using the same instance.

Example

```
// Connect to a server using a standard (non-secure) connection on
// the default port. The username and password are not encrypted.

bResult = ftpClient.Connect(lpszRemoteHost,
                            FTP_PORT_DEFAULT,
                            lpszUserName,
                            lpszPassword,
                            FTP_TIMEOUT,
                            FTP_OPTION_DEFAULT);

// Connect to a server using the default port and then initiate a
// secure connection using the AUTH TLS command

bResult = ftpClient.Connect(lpszRemoteHost,
                            FTP_PORT_DEFAULT,
                            lpszUserName,
                            lpszPassword,
                            FTP_TIMEOUT,
                            FTP_OPTION_PASSIVE | FTP_OPTION_SECURE_EXPLICIT);

// Connect to a server on port 990 and immediately initiate a
```

```
// secure connection as soon as the connection is established

bResult = ftpClient.Connect(lpszRemoteHost,
                            FTP_PORT_SECURE,
                            lpszUserName,
                            lpszPassword,
                            FTP_TIMEOUT,
                            FTP_OPTION_PASSIVE | FTP_OPTION_SECURE_IMPLICIT);

// Connect to a server on port 22 using the Secure Shell (SFTP)
// protocol to transfer files

bResult = ftpClient.Connect(lpszRemoteHost,
                            FTP_PORT_SSH,
                            lpszUserName,
                            lpszPassword,
                            FTP_TIMEOUT,
                            FTP_OPTION_SECURE_SHELL);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateSecurityCredentials](#), [Disconnect](#), [GetSecurityInformation](#), [ProxyConnect](#), [SetChannelMode](#)

CFtpClient::ConnectUrl Method

```
BOOL ConnectUrl(  
    LPCTSTR lpszURL,  
    UINT nTimeout,  
    DWORD dwOptions  
);
```

The **ConnectUrl** method establishes a connection with the specified server using a URL.

Parameters

lpszURL

A pointer to a string which specifies the URL for the server. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
FTP_OPTION_PASSIVE	This option specifies the client should attempt to establish the data connection with the server. When the client uploads or downloads a file, normally the server establishes a second connection back to the client which is used to transfer the file data. However, if the local system is behind a firewall or a NAT router, the server may not be able to create the data connection and the transfer will fail. By specifying this option, it forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_NOAUTH	This option specifies the server does not require authentication, or that it requires an alternate authentication method. When this option is used, the client connection is flagged as authenticated as

	<p>soon as the connection to the server has been established. Note that using this option to bypass authentication may result in subsequent errors when attempting to retrieve a directory listing or transfer a file. It is recommended that you consult the technical reference documentation for the server to determine its specific authentication requirements.</p>
FTP_OPTION_KEEPALIVE	<p>This option specifies the client should attempt to keep the connection with the server active for an extended period of time. It is important to note that regardless of this option, the server may still choose to disconnect client sessions that are holding the command channel open but are not performing file transfers.</p>
FTP_OPTION_NOAUTHRSA	<p>This option specifies that RSA authentication should not be used with SSH-1 connections. This option is ignored with SSH-2 connections and should only be specified if required by the server. This option has no effect on standard or secure connections using SSL.</p>
FTP_OPTION_NOPWDNUL	<p>This option specifies the user password cannot be terminated with a null character. This option is ignored with SSH-2 connections and should only be specified if required by the server. This option has no effect on standard or secure connections using SSL.</p>
FTP_OPTION_NOREKEY	<p>This option specifies the client should never attempt a repeat key exchange with the server. Some SSH servers do not support rekeying the session, and this can cause the client to become non-responsive or abort the connection after being connected for an hour. This option has no effect on standard or secure connections using SSL.</p>
FTP_OPTION_COMPATSID	<p>This compatibility option changes how the session ID is handled during public key authentication with older SSH servers. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions. This option has no effect on standard or secure connections using SSL.</p>
FTP_OPTION_COMPATHMAC	<p>This compatibility option changes how the HMAC authentication codes are generated. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions. This option has no effect on standard or secure connections using SSL.</p>

FTP_OPTION_VIRTUALHOST	This option specifies the server supports virtual hosting, where multiple domains are hosted by a server using the same external IP address. If this option is enabled, the client will send the HOST command to the server upon establishing a connection.
FTP_OPTION_VERIFY	This option specifies that file transfers should be automatically verified after the transfer has completed. If the server supports the XMD5 command, the transfer will be verified by calculating an MD5 hash of the file contents. If the server does not support the XMD5 command, but does support the XCRC command, the transfer will be verified by calculating a CRC32 checksum of the file contents. If neither the XMD5 or XCRC commands are supported, the transfer is verified by comparing the size of the file. Automatic file verification is only performed for binary mode transfers because of the end-of-line conversion that may occur when text files are uploaded or downloaded.
FTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. This option is the same as specifying FTP_OPTION_SECURE_IMPLICIT which immediately performs the SSL/TLS protocol negotiation when the connection is established.
FTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to immediately establish secure SSL/TLS connection with the server. This option is typically used when connecting to a server on port 990, which is the default port number used for FTPS.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should establish a standard connection to the server and then use the AUTH command to negotiate an explicit secure connection. This option is typically used when connecting to the server on ports other than 990.
FTP_OPTION_SECURE_SHELL	This option specifies the client should use the Secure Shell (SSH) protocol to establish the connection. This option will automatically be selected if the connection is established using port 22, the default port for SSH.
FTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility

	with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
FTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established. This option also forces all connections to be outbound and enables the firewall compatibility features in the client.
FTP_OPTION_KEEPALIVE_DATA	This option specifies the client should attempt to keep the control connection active during a file transfer. Normally, when a data transfer is in progress, no additional commands are issued on the control channel until the transfer completes. Specifying this option automatically enables the FTP_OPTION_KEEPALIVE option and forces the client to continue to issue NOOP commands during the file transfer. This option only applies to FTP and FTPS connections and has no effect on connections using SFTP (SSH).
FTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
FTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
FTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller files being uploaded or downloaded using fast network connections.
FTP_OPTION_TLS_REUSE	This option specifies that TLS session reuse should be enabled for secure connections. This option is only supported on Windows 8.1 or Windows Server 2012 R2 and later platforms, and it should only be used when explicitly required by the server. This

option is not compatible with servers built using OpenSSL 1.0.2 and earlier versions which do not provide Extended Master Secret (EMS) support as outlined in RFC7627.
--

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ConnectUrl** method is a high-level method that uses an FTP URL to establish a connection with a server. Unlike the other connection related methods such as **Connect**, this method does more than simply connect to the server. It will also authenticate the client session, change the current working directory and set the default file transfer mode. By default, this method will always place the client in passive mode, ensuring the broadest compatibility with most servers. The **ValidateUrl** method can be used to verify that a URL is valid prior to calling this function.

The URL must be complete, and specify either a standard or secure FTP scheme:

```
[ftp|ftps|sftp]://[username : password] @[remotehost] [:remoteport] /  
[path / ...] [filename]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. If a path is specified as part of the URL, the method will attempt to change the current working directory. The paths in an FTP URL are relative to the home directory of the user account and are not absolute paths starting at the root directory on the server. If a file name is also specified in the URL, it will be ignored and only the file path will be used. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "ftp" scheme is used and the FTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, either the "ftps" or "sftp" scheme must be specified.

The **ConnectUrl** method is designed to provide a simpler, more convenient interface to establishing a connection with a server. However, complex connections such as those using a proxy server or a secure connection which uses a client certificate will require the program to use the lower-level connection methods. If you only need to upload or download a file using a URL, then refer to the **UploadFile** and **DownloadFile** methods.

The *dwOptions* argument can be used to specify the threading model that is used by the class when a connection is established. By default, the client session is initially attached to the thread that created it. From that point on, until the connection is terminated, only the owner may invoke methods in that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the FTP_OPTION_FREETHREAD option enables any thread to call methods in any instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the class and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a method in the class, it must ensure that no other thread will call another method at the same time using the same instance.

Example

```
CFtpClient ftpClient;  
  
if (!ftpClient.ConnectUrl(_T("ftp://ftp.sockettools.com/pub/")))  
{
```

```
    ftpClient.ShowError();  
    return;  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [Disconnect](#), [DownloadFile](#), [UploadFile](#), [ValidateUrl](#)

CFtpClient::CreateDirectory Method

```
INT CreateDirectory(  
    LPCTSTR lpszDirectory  
);
```

The **CreateDirectory** method creates the specified directory on the server.

Parameters

lpszDirectory

Points to a string that specifies the name of the directory. The file pathing and name conventions must be that of the server.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method uses the MKD command to create the directory. The user must have the appropriate permission to create the specified directory.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ChangeDirectory](#), [GetDirectory](#), [RemoveDirectory](#)

CFtpClient::CreateSecurityCredentials Method

```
BOOL CreateSecurityCredentials(  
    DWORD dwProtocol,  
    DWORD dwOptions,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName  
);
```

```
BOOL CreateSecurityCredentials(  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName  
);
```

```
BOOL CreateSecurityCredentials(  
    LPCTSTR lpszCertName  
);
```

The **CreateSecurityCredentials** method establishes the security credentials for the client session.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols.

	This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.
SECURITY_PROTOCOL_SSH	Either version 1.0 or 2.0 of the Secure Shell protocol should be used when establishing the connection. The correct protocol is automatically selected based on the version of the protocol that is supported by the server.
SECURITY_PROTOCOL_SSH1	The Secure Shell 1.0 protocol should be used when establishing the connection. This is an older version of the protocol which should not be used unless explicitly required by the server. Most modern SSH server support version 2.0 of the protocol.
SECURITY_PROTOCOL_SSH2	The Secure Shell 2.0 protocol should be used when establishing the connection. This is the default version of the protocol that is supported by most SSH servers.

dwOptions

A value which specifies one or options. This value should always be zero for connections using SSH. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for

	the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A pointer to a string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A pointer to a string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A pointer to a string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the

function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Example

```
pClient->CreateSecurityCredentials(  
    SECURITY_PROTOCOL_DEFAULT,  
    0,  
    NULL,  
    NULL,  
    lpszCertStore,  
    lpszCertName);  
  
bConnected = pClient->Connect(lpszHostName,  
    FTP_PORT_SECURE,  
    FTP_TIMEOUT,  
    FTP_OPTION_SECURE);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [DeleteSecurityCredentials](#), [GetSecurityInformation](#), [SECURITYCREDENTIALS](#)

CFtpClient::DeleteFile Method

```
INT DeleteFile(  
    LPCTSTR lpszFileName  
);
```

The **DeleteFile** method deletes the specified file from the server.

Parameters

lpszFileName

Points to a string that specifies the name of the remote file to delete. The file pathing and name conventions must be that of the server.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The current user must have the appropriate permission to delete the file, or an error will be returned by the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFile](#), [PutFile](#), [RenameFile](#)

CFtpClient::DeleteSecurityCredentials Method

```
VOID DeleteSecurityCredentials();
```

The **DeleteSecurityCredentials** method releases the security credentials for the current session.

Parameters

None.

Return Value

None.

Remarks

This method can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated. The security credentials are released when the class destructor is called, so it is normally not required that the application explicitly call this method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateSecurityCredentials](#)

CFtpClient::DetachHandle Method

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

See Also

[AttachHandle](#), [GetHandle](#)

CFtpClient::DisableEvents Method

```
INT DisableEvents();
```

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the client has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[EnableEvents](#), [RegisterEvent](#)

CFtpClient::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[EnableTrace](#)

CFtpClient::Disconnect Method

VOID Disconnect();

The **Disconnect** method terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[Connect](#), [ProxyConnect](#)

CFtpClient::DownloadFile Method

```
BOOL DownloadFile(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszFileURL,  
    UINT nTimeout  
    DWORD dwOptions  
    LPFTPTRANSFERSTATUS lpStatus  
    FTPEVENTPROC lpEventProc  
    DWORD_PTR dwParam  
);
```

The **DownloadFile** method downloads the specified file from the server to the local system.

Parameters

lpszLocalFile

A pointer to a string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszFileURL

A pointer to a string that specifies the complete URL of the file to be downloaded. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
FTP_OPTION_PASSIVE	This option specifies the client should attempt to establish the data connection with the server. When the client uploads or downloads a file, normally the server establishes a second connection back to the client which is used to transfer the file data. However, if the local system is behind a firewall or a NAT router, the server may not be able to create the data connection and the transfer will fail. By specifying this option, it forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data

	connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should use the AUTH command to negotiate an explicit secure connection. Some servers may only require this when connecting to the server on ports other than 990.

lpStatus

A pointer to an FTPTRANSFERSTATUS structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **DownloadFile** method provides a convenient way for an application to download a file in a single method call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session, change the current working directory if necessary and then download the file to the local system. The URL must be complete, and specify either a standard or secure FTP scheme:

```
[ftp|ftps|sftp]://[username : password] @[remotehost] [:remoteport] /
[path / ...] [filename] [;type=a|i]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. If a path is specified as part of the URL, the method will attempt to change the current working directory. Note that the path in an FTP URL is relative to the home directory of the user account and is not an absolute path starting at the root directory on the server. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "ftp" scheme is used and the FTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, either the "ftps" or "sftp" scheme must be specified.

The optional "type" value at the end of the file name determines if the file should be downloaded as a text or binary file. A value of "a" specifies that the file should be downloaded as a text file. A value of "i" specifies that the file should be downloaded as a binary file. If the type is not explicitly specified, the file will be downloaded as a binary file.

The *lpStatus* parameter can be used by the application to determine the final status of the transfer, including the total number of bytes copied, the amount of time elapsed and other

information related to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The *lpEventProc* parameter specifies a pointer to a function which will be periodically called during the file transfer process. This can be used to check the status of the transfer by calling **GetTransferStatus** and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The *dwParam* parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the *this* pointer as the value, and then cast it back to an object pointer inside the callback function. If no event handler is required, then a NULL pointer can be specified as the value for *lpEventProc* and the *dwParam* parameter will be ignored.

The **DownloadFile** method is designed to provide a simpler interface for downloading a file. However, complex connections such as those using a proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **Connect** and then use **GetFile** to download the file.

Example

```
CFtpClient ftpClient;
CString strLocalFile = _T("c:\\temp\\database.mdb");
CString strFileURL = _T("ftp://ftp.example.com/updates/database.mdb");

// Download the file using the specified URL
if (!ftpClient.DownloadFile(strLocalFile, strFileURL))
{
    ftpClient.ShowError();
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpEventProc](#), [GetFile](#), [GetTransferStatus](#), [UploadFile](#), [FTPTRANSFERSTATUS](#)

CFtpClient::EnableEvents Method

```
INT EnableEvents(  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Constant	Description
FTP_EVENT_CONNECT	The control connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
FTP_EVENT_DISCONNECT	The server has closed the control connection to the client. The client should read any remaining data and disconnect.
FTP_EVENT_OPENFILE	The data connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
FTP_EVENT_CLOSEFILE	The server has closed the data connection to the client. The client should read any remaining data and close the data channel.
FTP_EVENT_READFILE	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of

	the data. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_WRITEFILE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
FTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and reconnect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
FTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
FTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.
FTP_EVENT_GETFILE	This event is generated when a file download has completed. If multiple files are being downloaded, this event will be generated for each file.
FTP_EVENT_PUTFILE	This event is generated when a file upload has completed. If multiple files are being uploaded, this event will be generated for each file.

It is not required that the client be placed in asynchronous mode in order to receive command and progress event notifications. To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[DisableEvents](#), [RegisterEvent](#)

CFtpClient::EnableFeature Method

```
BOOL EnableFeature(  
    DWORD dwFeature,  
    BOOL bEnable  
);
```

The **EnableFeature** method enables or disables a specific server feature available to the client.

Parameters

dwFeature

An unsigned integer which specifies the feature to be enabled to disabled for the current client session. Refer to the documentation for the **GetFeatures** method for a list of available features.

bEnabled

A boolean flag which specifies if the feature should be enabled or disabled. If the value is non-zero, the library will attempt to use that feature on the server. If the value is zero, the feature is disabled. If an application calls a method which requires a specific feature and that feature is disabled, the method will fail with an error indicating the feature is not supported.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return a value of zero. To get extended error information, call the **GetLastError** method.

Remarks

The **EnableFeature** method is used to enable or disable a specific feature for the current session. When a client connection is first established, features are enabled based on the server type and the server's response to the FEAT command. However, as the client issues commands to the server, if the server reports that the command is unrecognized that feature will automatically be disabled in the client. An application can use the **EnableFeature** method to control what commands will be sent to the server, or re-enable a command that was previously disabled.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[GetFeatures](#), [SetFeatures](#)

CFtpClient::EnableTrace Method

```
BOOL EnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **EnableTrace** method enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace method logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableTrace](#)

CFtpClient::EnumFiles Method

```
INT EnumFiles(  
    LPCTSTR lpszDirectory,  
    LPCTSTR lpszFileMask,  
    DWORD dwOptions,  
    LPFTPFILESTATUSEX lpFileList,  
    INT nMaxFiles  
);
```

The **EnumFiles** method populates an array of structures that contain information about the files in a directory.

Parameters

lpszDirectory

A pointer to a string that specifies the name of a directory on the server. If this parameter is NULL or points to an empty string, the method will return the files in the current working directory. This string cannot contain wildcard characters and must specify a valid directory name that exists on the server.

lpszFileMask

A pointer to a string that specifies a wildcard file mask that is used to return a subset of files in the directory. If this parameter is NULL or an empty string then all of the files in the directory will be returned.

dwOptions

An unsigned integer value that specifies one or more options. This parameter can be a combination of one or more of the following values:

Constant	Description
FTP_ENUM_DEFAULT	The method will return both regular files and subdirectories.
FTP_ENUM_FILE	The method will return only regular files.
FTP_ENUM_DIRECTORY	The method will return only subdirectories.
FTP_ENUM_FULLPATH	The method will return the full path of the file or subdirectory.

lpFileList

A pointer to an array of [FTPFILESTATUSEX](#) structures which contains information about each of the files in the specified directory. This parameter cannot be NULL, and the array must be large enough to store the number of files specified by the *nMaxFiles* parameter.

nMaxFiles

An integer value that specifies the maximum number of files that should be returned. This value must be greater than zero and the *lpFileList* parameter must provide an array that is large enough to store information about each file.

Return Value

If the method succeeds, the return value is the number of files returned by the method. If the directory is empty or there are no files that match the specified wildcard file mask, the method will return zero. If the method fails, the return value is `FTP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **EnumFiles** method provides a high-level interface for obtaining a list of available files in a directory on the server in a single function call. This is an alternative to opening a directory and returning information about each file by calling the **GetNextFile** method in a loop.

This method temporarily changes the current working directory to the directory specified by the *lpzDirectory* parameter. The current working directory will be restored to its original value when the method returns. The user must have the appropriate permissions to access the directory or this method will fail.

To obtain information on a subset of files in the directory, you can specify a wildcard file mask. For FTP and FTPS (SSL) sessions, this value is passed as a parameter to the LIST command and the server performs the wildcard matching. For SFTP (SSH) sessions the wildcard matching is performed by the library, and the standard conventions for Windows file wildcards are used.

By default, the *szFileName* member for each **FTPFILESTATUSEX** structure will contain the base file name. If the FTP_ENUM_FULLPATH option is specified, the method will return the full path name to the file. The library must be able to automatically determine the path delimiter that is used by the server. This is done by examining how the server identifies itself, the current directory format and the path the server returns for the current working directory. For example, UNIX based servers use the forward slash as a path delimiter. If the method cannot determine what the appropriate path delimiter is, it will ignore this option and return only the base file name.

This method will cause the current thread to block until the file listing completes, a timeout occurs or the operation is canceled.

Example

```
LPFTPFILESTATUSEX lpFileList = new FTPFILESTATUSEX[MAXFILECOUNT];

// Return all of the regular files in the current working directory
INT nResult = pClient->EnumFiles(FTP_ENUM_FILE, lpFileList, MAXFILECOUNT);

if (nResult == FTP_ERROR)
{
    DWORD dwError = pClient->GetLastError();
    _tprintf(_T("EnumFiles failed, error 0x%08lx\n"), dwError);
    return;
}

_tprintf(_T("EnumFiles returned %d files\n"), nResult);
for (INT nIndex = 0; nIndex < nResult; nIndex++)
{
    _tprintf(_T("file=\"%s\" size=%I64d\n"), lpFileList[nIndex].szFileName,
            lpFileList[nIndex].uiFileSize);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFileStatus](#), [GetFirstFile](#), [GetNextFile](#), [OpenDirectory](#)

CFtpClient::FreezeEvents Method

```
INT FreezeEvents(  
    BOOL bFreeze  
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the client.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableEvents](#), [EnableEvents](#), [RegisterEvent](#)

CFtpClient::GetActivePorts Method

```
INT GetActivePorts(  
    UINT * lpnLowPort,  
    UINT * lpnHighPort  
);
```

The **GetActivePorts** method returns the local port numbers used for active mode file transfers.

Parameters

lpnLowPort

Points to an unsigned integer that will contain the low port number when the function returns.

lpnHighPort

Points to an unsigned integer that will contain the high port number when the function returns.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method is used to determine the default port numbers being used for active mode file transfers. When using active mode, the client listens for an inbound connection from the server rather than establishing an outbound connection for the data transfer. In most cases, passive mode transfers are preferred because they mitigate potential compatibility issues with firewalls and NAT routers.

If active mode transfers are required, the default port range used when listening for the server connection is between 1024 and 5000. This is the standard range of ephemeral ports used by the Windows operating system. However, under some circumstances that range of ports may be too small, or a firewall may be configured to deny inbound connections on ephemeral ports. In that case, the **SetActivePorts** method can be used to specify a different range of port numbers.

While it is technically permissible to assign the low and high port numbers to the same value, effectively specifying a single active port number, this is not recommended as it can cause the transfer to fail unexpectedly if multiple file transfers are performed. A minimum range of at least 1000 ports is recommended. For example, if you specify a low port value of 40000 then it is recommended that the high port value be at least 41000. The maximum port value is 65535.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SetActivePorts](#), [SetPassiveMode](#)

CFtpClient::GetBufferSize Method

```
INT GetBufferSize();
```

The **GetBufferSize** method returns the size in bytes of an internal buffer that will be used during data transfers.

Parameters

None.

Return Value

If the method succeeds, the return value is the size of the internal buffer that will be used in data transfers. If the method fails, the return value is `FTP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The speed of data transfers, particularly on uploads, may be sensitive to network type and configuration, and the size of the internal buffer used for data transfers. The default size of this buffer will result in good performance for a wide range of network characteristics. A larger buffer will not necessarily result in better performance. For example, a value of 1460, which is the typical Maximum Transmission Unit (MTU), may be optimal in many situations.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

See Also

[SetBufferSize](#)

CFtpClient::GetChannelMode Method

```
INT GetChannelMode(  
    INT nChannel  
);
```

The **GetChannelMode** method returns the mode of the specified communications channel.

Parameters

nChannel

An integer value which specifies which channel to return information for. It may be one of the following values:

Constant	Description
FTP_CHANNEL_COMMAND	Return information about the command channel. This is the communication channel used to send commands to the server and receive command result and status information from the server.
FTP_CHANNEL_DATA	Return information about the data channel. This is the communication channel used to send or receive data during a file transfer.

Return Value

If the method succeeds, the return value is the mode for the specified channel. If the method fails, it will return FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetChannelMode** method returns a integer which may be one of the following values:

Constant	Description
FTP_CHANNEL_CLEAR	The channel is not encrypted. This is the default mode for both channels when a standard, non-secure connection is established with the server.
FTP_CHANNEL_SECURE	The channel is encrypted. This is the default mode for both channels when a secure connection is established with the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

See Also

[SetChannelMode](#)

CFtpClient::GetClientQuota Method

```
INT GetClientQuota(  
    LPFTPCLIENTQUOTA LpClientQuota  
);
```

The **GetClientQuota** method returns information about file quotas for the current client session.

Parameters

LpClientQuota

A pointer to an [FTPCLIENTQUOTA](#) structure which contains the quota information returned by the server.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is `FTP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

This method uses the `XQUOTA` command to obtain information for the current client session. If the server does not support this command, the method will fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

See Also

[GetServerInformation](#), [GetServerType](#)

CFtpClient::GetData Method

```
INT GetData(  
    LPCTSTR lpszRemoteFile,  
    LPBYTE lpBuffer,  
    LPDWORD lpdwLength  
);
```

```
INT GetData(  
    LPCTSTR lpszRemoteFile,  
    HGLOBAL* lpBuffer,  
    LPDWORD lpdwLength  
);
```

```
INT GetData(  
    LPCTSTR lpszRemoteFile,  
    LPTSTR lpBuffer,  
    DWORD dwMaxLength  
);
```

```
INT GetData(  
    LPCTSTR lpszRemoteFile,  
    CString& strBuffer  
);
```

The **GetData** method transfers the contents of a file on the server to the specified buffer.

Parameters

lpszRemoteFile

A pointer to a string that specifies the file on the server that will be transferred to the local system. The file naming conventions must be that of the host operating system.

lpBuffer

A pointer to a buffer which will contain the data transferred from the server. In alternate forms of the method, this argument may also be a pointer to a global memory handle or reference a **CString** object.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpBuffer* parameter. If the *lpBuffer* parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual length of the file that was downloaded.

dwMaxLength

An unsigned integer value which specifies the maximum number of characters can be copied into the *lpBuffer* string. This parameter is used with the version of the method that returns the data in a character array and includes the terminating null character. This value must be greater than one and the *lpBuffer* parameter cannot be NULL, otherwise the method will return an error.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is **FTP_ERROR**. To get extended error information, call **GetLastError**.

Remarks

The **GetData** method is used to download the contents of a remote file into a local buffer. The method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the *lpBuffer* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpBuffer* parameter point to a global memory handle which will contain the file data when the method returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. See the example code below.

If the third method or fourth method is used, where the data is returned in a string buffer, the data may be modified so that the end-of-line character sequence matches the convention used by the Windows platform (a carriage return character followed by a linefeed). If Unicode is being used, the data will be converted from a byte array to a Unicode string. An application should only use these versions of the **GetData** method if the remote file contains text.

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Example

```
HGLOBAL hglbBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Return the file data into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hglbBuffer parameter
nResult = pClient->GetData(lpszRemoteFile, &hglbBuffer, &cbBuffer);

if (nResult != FTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // contents of the file data
    lpBuffer = (LPBYTE)GlobalLock(hglbBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hglbBuffer);
    GlobalFree(hglbBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ChangeDirectory](#), [EnableEvents](#), [GetFile](#), [GetTransferStatus](#), [PutData](#), [PutFile](#), [RegisterEvent](#), [SetBufferSize](#)

CFtpClient::GetDirectory Method

```
INT GetDirectory(  
    LPTSTR lpszDirectory,  
    INT cbDirectory  
);
```

The **GetDirectory** method copies the current working directory on the server to the specified buffer.

Parameters

lpszDirectory

Points to a buffer that will contain the name of the current working directory on the server. The file pathing and name conventions must be that of the server.

cbDirectory

The maximum number of characters that may be copied into the buffer, including the terminating null-character.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method sends the PWD command to the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ChangeDirectory](#), [CreateDirectory](#), [RemoveDirectory](#)

CFtpClient::GetDirectoryFormat Method

INT GetDirectoryFormat();

The **GetDirectoryFormat** method returns an identifier which specifies what format is being used by the server to list files. By default, the library will automatically determine the appropriate format, but this value may be overridden by the **SetDirectoryFormat** method.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

Refer to the **SetDirectoryFormat** method for a list of directory format types that are supported by the library. This method can be used to determine which format was selected by the library after a file listing has been retrieved.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[CloseDirectory](#), [GetFileStatus](#), [GetFirstFile](#), [GetNextFile](#), [OpenDirectory](#), [SetDirectoryFormat](#)

CFtpClient::GetErrorString Method

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);  
  
INT GetErrorString(  
    DWORD dwErrorCode,  
    CString& strDescription  
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLastError](#), [SetLastError](#), [ShowError](#)

CFtpClient::GetFeatures Method

DWORD GetFeatures();

The **GetFeatures** method returns the server features available to the client.

Parameters

None.

Return Value

If the method succeeds, the return value is one or more bit flags which specify the features that are available to the client. If the method fails, it will return zero. Because it is possible that no features would be enabled, a return value of zero does not always indicate an error. An application should call **GetLastError** to determine if an error code has been set.

Remarks

The **GetFeatures** method returns a value which may be a combination of one or more of the following bit flags:

Constant	Description
FTP_FEATURE_SIZE	The server supports the SIZE command to determine the size of a file. If this feature is not enabled, the library will attempt to use the MLST or STAT command to determine the file size.
FTP_FEATURE_STAT	The server supports using the STAT command to return information about a specific file. If this feature is not enabled, the client may not be able to obtain information about a specific file such as its size, permissions or modification time.
FTP_FEATURE_MDTM	The server supports the MDTM command to obtain information about the modification time for a specific file. This command may also be used to set the file time on the server.
FTP_FEATURE_REST	The server supports restarting file transfers using the REST command. If this feature is not enabled, the client will not be able to restart file transfers and must upload or download the complete file.
FTP_FEATURE_SITE	The server supports site specific commands using the SITE command. If this feature is not enabled, no site specific commands will be sent to the server.
FTP_FEATURE_IDLE	The server supports setting the idle timeout period using the SITE IDLE command to specify the number of seconds that the client may idle before the server terminates the connection.
FTP_FEATURE_CHMOD	The server supports modifying the permissions of a specific file using the SITE CHMOD command. If this feature is not enabled, the client will not be able to set the permissions for a file.

FTP_FEATURE_AUTH	The server supports explicit SSL sessions using the AUTH command. If this feature is not enabled, the client will only be able to connect to a secure server that uses implicit SSL connections. Changing this feature has no effect on standard, non-secure connections.
FTP_FEATURE_PBSZ	The server supports the PBSZ command which specifies the buffer size used with secure data connections. If this feature is disabled, it may prevent the client from changing the protection level on the data channel. Changing this feature has no effect on standard, non-secure connections.
FTP_FEATURE_PROT	The server supports the PROT command which specifies the protection level for the data channel. If this feature is disabled, the client will be unable to change the protection level on the data channel. Changing this feature has no effect on standard, non-secure connections.
FTP_FEATURE_CCC	The server supports the CCC command which returns the command channel to a non-secure mode. Changing this feature has no effect on standard, non-secure connections.
FTP_FEATURE_HOST	The server supports the HOST command which enables a client to specify the hostname after establishing a connection with a server that supports virtual hosting.
FTP_FEATURE_MLST	The server supports the MLST command which returns status information for files. If this feature is enabled, the MLST command will be used instead of the STAT command.
FTP_FEATURE_MFMT	The server supports the MFMT command which is used to change the last modification time for a file. If this command is supported, it is used instead of the MDTM command to change the modification time for a file.
FTP_FEATURE_XCRC	The server supports the XCRC command which returns the CRC-32 checksum for the contents of a specified file. This command is used for file verification.
FTP_FEATURE_XMD5	The server supports the XMD5 command which returns an MD5 hash for the contents of a specified file. This command is used for file verification.
FTP_FEATURE_LANG	The server supports the LANG command which sets the language used for the current client session. Command responses and file naming conventions will use the specified language.
FTP_FEATURE_UTF8	The server supports the OPTS UTF-8 command which specifies UTF-8 encoding when specifying filenames. This feature is typically used in conjunction with setting the default language for the client session.
FTP_FEATURE_XQUOTA	The server supports the XQUOTA command which returns quota information for the current client session.
FTP_FEATURE_UTIME	The server supports the UTIME command which is used to

change the last modification time for a specified file.

When a client connection is first established, features are enabled based on the server type and the server's response to the FEAT command. However, as the client issues commands to the server, if the server reports that the command is unrecognized that feature will automatically be disabled in the client.

For example, the first time an application calls the **GetFileSize** method to determine the size of a file, the library will try to use the SIZE command. If the server reports that the SIZE command is not available, that feature will be disabled and the library will not use the command again during the session unless it is explicitly re-enabled. This is designed to prevent the library from repeatedly sending invalid commands to a server, which may result in the server aborting the connection.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[EnableFeature](#), [SetFeatures](#)

CFtpClient::GetFile Method

```
INT GetFile(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszRemoteFile,  
    DWORD dwOptions  
    DWORD dwOffset  
);
```

The **GetFile** method transfers the specified file on the server to the local system.

Parameters

lpszLocalFile

A pointer to a string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A pointer to a string that specifies the file on the server that will be transferred to the local system. The file naming conventions must be that of the host operating system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Constant	Description
FTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the local file exists, it will be overwritten with the contents of the downloaded file.
FTP_TRANSFER_APPEND	This option specifies that if the local file exists, the contents of file on the server is appended to the local file. If the local file does not exist, it is created.

dwOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ChangeDirectory](#), [EnableEvents](#), [GetMultipleFiles](#), [GetTransferStatus](#), [PutFile](#), [PutMultipleFiles](#), [RegisterEvent](#), [SetBufferSize](#), [VerifyFile](#)

CFtpClient::GetFileList Method

```
INT GetFileList(  
    LPCTSTR lpszDirectory,  
    DWORD dwOptions,  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);
```

```
INT GetFileList(  
    LPCTSTR lpszDirectory,  
    DWORD dwOptions,  
    CString& strBuffer  
);
```

The **GetFileList** method returns an unparsed list of files in the specified directory.

Parameters

lpszDirectory

A pointer to a string that specifies the name of a directory and/or a wildcard file mask. The format of the directory name must match the file naming conventions of the server. If this parameter is NULL or points to an empty string, the current working directory will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Constant	Description
FTP_LIST_DEFAULT	This option specifies the server should return a complete listing of files in the specified directory with as much detail as possible. This typically means that the file size, date, ownership and access rights will be returned to the client. Information about the files are returned in lines of text, with each line terminated by carriage return and linefeed (CRLF) characters. The exact format of the data returned is specific to the server operating system.
FTP_LIST_NAMEONLY	This option specifies the server should only return a list of file names, with no additional information about the file. Each file name is terminated by carriage return and linefeed (CRLF) characters.

lpszBuffer

A pointer to a string buffer that will contain the list of files when the function returns. This buffer should be large enough to store the complete file listing and a terminating null character. If the buffer is smaller than the total amount of data returned by the server, the data will be truncated. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the number of bytes copied into the string buffer, not including the terminating null character. If the function fails, the return value is `FTP_ERROR`. To get extended error information, call **FtpGetLastError**.

Remarks

The **GetFileList** method returns a list of files in the specified directory, copying the data to a string buffer. Unlike the other methods like **EnumFiles** that parse a directory listing and return information in an **FTPFILESTATUS** structure, this method returns the unparsed file list data. The actual format of the data that is returned depends on the operating system and how the server implements file listings. For example, UNIX servers typically return the output from the `/bin/ls` command.

Some servers may not support file listings for any directory other than the current working directory. If an error is returned when specifying a directory name, try changing the current working directory using the **ChangeDirectory** method and then call this method again, passing `NULL` or an empty string as the *lpszDirectory* parameter.

This method can be particularly useful when the client is connected to a server that returns file listings in a format that is not recognized by the library. The application can retrieve the unparsed file listing from the server and parse the contents. Note that if you specify the `FTP_LIST_NAMEONLY` option, the data will only contain a list of file names and there will be no way for the application to know if they represent a regular file or a subdirectory.

This method is supported for both FTP and SFTP (SSH) connections, however the format of the data may differ depending on which protocol is used. Most UNIX based FTP servers will not list files and subdirectories that begin with a period, however most SFTP servers will return a list of all files, even those that begin with a period.

This function will cause the current thread to block until the file listing completes, a timeout occurs or the operation is canceled.

Example

```
CString strFileList;

nResult = pClient->GetFileList(NULL, FTP_LIST_DEFAULT, strFileList);

if (nResult != FTP_ERROR)
    pEditCtl->SetWindowText(strFileList);
else
{
    pClient->ShowError();
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CloseDirectory](#), [EnumFiles](#), [GetDirectoryFormat](#), [GetFirstFile](#), [OpenDirectory](#), [GetNextFile](#)

CFtpClient::GetFileNameEncoding Method

```
INT GetFileNameEncoding();
```

The **GetFileNameEncoding** method returns an identifier which specifies what type of encoding is being used when file names are sent to the server.

Parameters

None.

Return Value

If the method succeeds, the return value is the type of encoding that is used. If the method fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

Refer to the **SetFileNameEncoding** method for a list of encoding types that are supported by the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[Command](#), [EnableFeature](#) [GetFeatures](#), [SetFileNameEncoding](#), [SetFeatures](#)

CFtpClient::GetFilePermissions Method

```
INT GetFilePermissions(  
    LPCTSTR lpzFileName,  
    LPDWORD lpdwPermissions  
);
```

The **GetFilePermissions** method returns information about the access permissions for a specific file on the server.

Parameters

lpzFileName

A pointer to a string which contains the name of the file that the access permissions are to be returned for. The filename cannot contain any wildcard characters.

lpdwPermissions

A pointer to an unsigned integer which will contain the access permissions for the file when the method returns. The file permissions are represented as bit flags, and may be one or more of the following values:

Constant	Description
FILE_OWNER_READ	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.
FILE_OWNER_WRITE	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_OWNER_EXECUTE	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_GROUP_READ	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
FILE_GROUP_WRITE	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_GROUP_EXECUTE	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_WORLD_READ	All users have permission to open the file for reading. This permission grants any user the right to download the file to

	the local system.
FILE_WORLD_WRITE	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.
FILE_WORLD_EXECUTE	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.

Return Value

If the method succeeds, the return value is a result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method uses the STAT command to retrieve information about the specified file. On some systems, the STAT command will not return information on files that contain spaces or tabs in the filename. In this case, the method will fail and value pointed to by the *lpdwPermissions* parameter will be zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFileStatus](#), [SetFilePermissions](#)

CFtpClient::GetFileSize Method

```
INT GetFileSize(  
    LPCTSTR lpszFileName,  
    LPDWORD lpdwFileSize  
);
```

The **GetFileSize** method returns the size of the specified file on the server.

Parameters

lpszFileName

Points to a string that specifies the name of the remote file.

lpdwFileSize

Points to an unsigned integer that will contain the size of the specified file in bytes.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method uses the SIZE command to determine the length of the specified file. Not all servers implement this command, in which case the method call will fail, and the *lpdwFileSize* parameter will be set to zero.

Note that if the file on the server is a text file, it is possible that the value returned by this method will not match the size of the file when it is downloaded to the local system. This is because different operating systems use different sequences of characters to mark the end of a line of text, and when a file is transferred in text mode, the end of line character sequence is automatically converted to a carriage return-linefeed, which is the convention used by the Windows platform.

Some FTP servers will refuse to return the size of a file if the current file type is set to FILE_TYPE_ASCII because the size of a text file on the server may not accurately reflect what the size of the file will be on the local system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFileStatus](#), [OpenDirectory](#)

CFtpClient::GetFileStatus Method

```
BOOL GetFileStatus(  
    LPCTSTR lpzFileName,  
    LPFTPFILESTATUS lpFileStatus  
);
```

The **GetFileStatus** method returns information about a specific file on the server.

Parameters

lpzFileName

A pointer to a string which contains the name of the file that status information will be returned on. The file name cannot contain any wildcard characters.

lpFileStatus

A pointer to an [FTPFILESTATUS](#) structure which contains information about the file returned by the server.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method uses the STAT command to retrieve information about the specified file. Unlike the **GetFirstFile** and **GetNextFile** methods, which read through a file list returned on the data channel, this method reads the result of a command string. For applications that need information about a specific file, using this method can be considerably faster than iterating through all of the files in a given directory. Note that not all servers support using the command in this way.

On some systems, the STAT command will not return information on files that contain spaces or tabs in the filename. In this case, the FTPFILESTATUS structure members will be empty strings and zero values.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CloseDirectory](#), [GetDirectoryFormat](#), [GetFirstFile](#), [GetNextFile](#), [GetTransferStatus](#), [OpenDirectory](#), [SetDirectoryFormat](#)

CFtpClient::GetFileTime Method

```
INT GetFileTime(  
    LPCTSTR lpszFileName,  
    LPSYSTEMTIME lpFileTime,  
    BOOL bLocalize  
);
```

The **GetFileTime** method returns the modification time for the specified file on the server.

Parameters

lpszFileName

Points to a string that specifies the name of the remote file.

lpFileTime

Points to a [SYSTEMTIME](#) structure that will be set to the current modification time for the remote file.

bLocalize

A boolean flag which specifies if the file time is localized to the current timezone. If this value is non-zero, then the file time is adjusted to that the time is local to the current system. If this value is zero, the file time is returned in UTC time.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is `FTP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **GetFileTime** method can be used to determine the date and time that a file was last modified on the server. The time may either be localized to the current system, or it may be returned as UTC time. If you plan on changing the values returned in the `SYSTEMTIME` structure and then calling **SetFileTime** method to modify the file time on the server, you should do not localize the time.

This method uses the `MDTM` command to determine the modification time of the specified file. If the server does not support this command, the method will attempt to use the `STAT` command to determine the file modification time.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFileStatus](#), [OpenDirectory](#), [SetFileTime](#)

CFtpClient::GetFileType Method

```
UINT GetFileType();
```

The **GetFileType** method returns the default file type for the current client session.

Parameters

None.

Return Value

If the method succeeds, the return value is an integer value that identifies the current file type. If the method fails, the return value is `FTP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **GetFileType** method will return one of the following values:

Value	Description
<code>FILE_TYPE_AUTO</code>	The file type should be automatically determined based on the file name extension. If the file extension is unknown, the file type should be determined based on the contents of the file. The library has an internal list of common text file extensions, and additional file extensions can be registered using the RegisterFileType method.
<code>FILE_TYPE_ASCII</code>	The file is a text file using the ASCII character set. For those servers which mark the end of a line with characters other than a carriage return and linefeed, it will be converted to the native client format. This is the file type used for directory listings.
<code>FILE_TYPE_EBCDIC</code>	The file is a text file using the EBCDIC character set. Local files will be converted to EBCDIC when sent to the server. Remote files will be converted to the native ASCII character set when retrieved from the server.
<code>FILE_TYPE_IMAGE</code>	The file is a binary file and no data conversion of any type is performed on the file. This is the default file type for most data files and executables. If the type of file cannot be automatically determined, it will always be considered a binary file.

If this method is called when connected to an SFTP (SSH) server, the default file type will always be `FILE_TYPE_IMAGE` because SFTP does not differentiate between text files and binary files.

The **SetFileType** function can be used to change the default file type.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

See Also

[OpenFile](#), [RegisterFileType](#), [SetFileMode](#), [SetFileStructure](#), [SetFileType](#), [SetPassiveMode](#)

CFtpClient::GetFirstFile Method

```
BOOL GetFirstFile(  
    LPFTPFILESTATUS lpFileStatus  
);
```

The **GetFirstFile** method returns the first file in the directory listing returned by the server after a call to the **OpenDirectory** method.

Parameters

lpFileStatus

A pointer to an [FTPFILESTATUS](#) structure which contains information about the file returned by the server.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This file list information returned by the server is cached by the library, allowing you to use this method to reposition back to the beginning of the file list.

Example

```
if (pClient->OpenDirectory() != FTP_ERROR)  
{  
    FTPFILESTATUS ftpFile;  
    BOOL bResult;  
  
    bResult = pClient->GetFirstFile(&ftpFile);  
    while (bResult)  
    {  
        // The ftpFile structure contains information about the file  
        bResult = pClient->GetNextFile(&ftpFile);  
    }  
  
    pClient->CloseDirectory();  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CloseDirectory](#), [GetDirectoryFormat](#), [GetFileStatus](#), [GetNextFile](#), [OpenDirectory](#), [SetDirectoryFormat](#)

CFtpClient::GetHandle Method

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

See Also

[AttachHandle](#), [DetachHandle](#), [IsInitialized](#)

CFtpClient::GetLastError Method

```
DWORD GetLastError();  
  
DWORD GetLastError(  
    CString& strDescription  
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError(0)** when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or FTP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

See Also

[GetErrorString](#), [SetLastError](#), [ShowError](#)

CFtpClient::GetMultipleFiles Method

```
INT GetMultipleFiles(  
    LPCTSTR lpszLocalDirectory,  
    LPCTSTR lpszRemoteDirectory,  
    LPCTSTR lpszFileMask  
);
```

The **GetMultipleFiles** method copies one or more files from the server to the local host, using the specified wildcard.

Parameters

lpszLocalDirectory

Pointer to a string which specifies the local directory where the files will be copied to. A NULL pointer or empty string specifies that files should be copied to the current working directory.

lpszRemoteDirectory

Pointer to a string which specifies the remote directory where the files will be copied from. A NULL pointer or empty string specifies that the files should be copied from the current working directory on the server.

lpszFileMask

Pointer to a string which specifies the files that are to be copied from the server to the local system. The file mask should follow the native conventions used for wildcard file matches on the server.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMultipleFiles** method is used to transfer files from the server to the local host which match a specified wildcard file mask. This method requires that the client be able to automatically list and parse directory listings from the server, otherwise an error will be returned. All files will be transferred using the current file type as specified by the **SetFileType** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ChangeDirectory](#), [GetFile](#), [PutFile](#), [PutMultipleFiles](#), [SetFileType](#)

CFtpClient::GetNextFile Method

```
BOOL GetNextFile(  
    LPFTPFILESTATUS lpFileStatus  
);
```

The **GetNextFile** method returns the next file in the directory listing returned by the server.

Parameters

lpFileStatus

A pointer to an [FTPFILESTATUS](#) structure which contains information about the file returned by the server.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetNextFile** method returns the next file in the directory listing. If the last file has been returned, the method will return zero and the client should call the **CloseDirectory** method to close the directory.

Example

```
if (pClient->OpenDirectory() != FTP_ERROR)  
{  
    FTPFILESTATUS ftpFile;  
    BOOL bResult;  
  
    bResult = pClient->GetFirstFile(&ftpFile);  
    while (bResult)  
    {  
        // The ftpFile structure contains information about the file  
        bResult = pClient->GetNextFile(&ftpFile);  
    }  
  
    pClient->CloseDirectory();  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CloseDirectory](#), [GetDirectoryFormat](#), [GetFileStatus](#), [GetFirstFile](#), [OpenDirectory](#), [SetDirectoryFormat](#)

CFtpClient::GetPriority Method

```
INT GetPriority();
```

The **GetPriority** method returns a value which specifies the priority of file transfers.

Parameters

None.

Return Value

If the method succeeds, the return value is the current file transfer priority. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetPriority** method can be used to determine the current priority assigned to file transfers performed by the client. It may be one of the following values:

Constant	Description
FTP_PRIORITY_NORMAL	The default priority which balances resource utilization and transfer speed. It is recommended that most applications use this priority.
FTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the transfer. It is typically used with worker threads running in the background when the amount of time required perform the transfer is not critical.
FTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the transfer and meters the bandwidth allocated for the transfer. This priority will increase the average amount of time required to complete a file transfer.
FTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the transfer, allocating more memory for internal buffering. It can be used when it is important to transfer the file quickly, and there are no other threads currently performing file transfers at the time.
FTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization while attempting to transfer the file as quickly as possible. If the file transfer is being performed in the main UI thread, this priority can cause the application to appear to become non-responsive. No events will be generated during the transfer.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

CFtpClient::GetProxyType Method

```
INT GetProxyType();
```

The **GetProxyType** method returns the type of proxy that the client is connected to. By default, no proxy server is specified and this method returns a value of FTP_PROXY_NONE. For a list of possible proxy server types, refer to the **ProxyConnect** method.

Parameters

None.

Return Value

If the method succeeds, the return value identifies the type of proxy that the client is connected to. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [ProxyConnect](#)

CFtpClient::GetResultCode Method

```
INT GetResultCode();
```

The **GetResultCode** method reads the result code returned by the server in response to a command. The result code is a three-digit numeric code, and indicates if the operation succeeded, failed or requires additional action by the client.

Parameters

None.

Return Value

If the method succeeds, the return value is the result code. If the method fails, the return value is `FTP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

Result codes are three-digit numeric values returned by the server. They may be broken down into the following ranges:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

See Also

[Command](#), [GetResultString](#)

CFtpClient::GetResultString Method

```
INT GetResultString(  
    LPTSTR lpszResult,  
    INT cbResult  
);  
  
INT GetResultString(  
    CString& strResult  
);
```

The **GetResultString** method returns the last message sent by the server along with the result code.

Parameters

lpszResult

A pointer to the buffer that will contain the result string returned by the server. An alternate form of the method accepts a **CString** argument which will contain the result string returned by the server.

cbResult

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server. If the method fails, the return value is `FTP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **GetResultString** method is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **GetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Command](#), [GetResultCode](#)

CFtpClient::GetSecurityInformation Method

```
BOOL GetSecurityInformation(  
    LPSECURITYINFO LpSecurityInfo  
);
```

The **GetSecurityInformation** method returns security protocol, encryption and certificate information about the current client connection.

Parameters

LpSecurityInfo

A pointer to a [SECURITYINFO](#) structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;  
securityInfo.dwSize = sizeof(SECURITYINFO);  
  
if (pClient->GetSecurityInformation(&securityInfo))  
{  
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)  
    {  
        MessageBox(NULL, _T("The connection is not secure"),  
                    _T("Connection"), MB_OK);  
    }  
    else  
    {  
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)  
        {  
            MessageBox(NULL, _T("The connection is secure"),  
                        _T("Connection"), MB_OK);  
        }  
        else  
        {  
            MessageBox(NULL, _T("The server certificate not valid"),  
                        _T("Connection"), MB_OK);  
        }  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [CreateSecurityCredentials](#), [SECURITYINFO](#)

CFtpClient::GetServerInformation Method

```
INT GetServerInformation(  
    LPTSTR lpszSystemInfo,  
    INT nMaxLength  
);
```

The **GetServerInformation** method returns information about the server, typically including the operating system type, version and platform.

Parameters

lpszSystemInfo

A pointer to the buffer that will contain the system information returned by the server.

nMaxLength

The maximum number of characters that can be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method sends the SYST command to the server. The first word will identify the type of operating system. The format for the remaining information depends on the server type. Typically it is a description of the operating system version and hardware platform.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [GetClientQuota](#), [GetServerStatus](#), [GetServerTimeZone](#), [GetServerType](#)

CFtpClient::GetServerStatus Method

```
INT GetServerStatus(  
    LPTSTR lpszStatus,  
    INT nMaxLength  
);  
  
INT GetServerStatus(  
    CString& strStatus  
);
```

The **GetServerStatus** method requests that the server return status information about itself.

Parameters

lpszStatus

A pointer to the buffer that will contain the system status returned by the server. The alternate form of this method also accepts a **CString** object which will contain the system status string.

nMaxLength

The maximum number of characters that can be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method sends the STAT command to the server. The format for the information returned depends on the server type. Typically it is a description of the server platform, version, current user and file transfer options.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetServerInformation](#), [GetServerType](#)

CFtpClient::GetServerTimeZone Method

```
INT GetServerTimeZone(  
    LPLONG lpnTimeZone  
);
```

The **GetServerTimeZone** method returns the timezone for the current server.

Parameters

lpnTimeZone

A pointer to a signed long integer which will contain the timezone offset in seconds.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This method sends the SITE ZONE command to the server to determine its timezone. The value returned is expressed as the number of seconds offset from Coordinated Universal Time (UTC). A positive value specifies a time west of UTC, while a negative value specifies a time east of UTC. For example, a value of 28800 would specify an offset of 8 hours west of UTC, which is the Pacific timezone.

The SIZE ZONE command is an extension that is not supported by all servers. If the server timezone cannot be determined, the method will fail and the value pointed to by the *lpnTimeZone* parameter will be zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

See Also

[GetClientQuota](#), [GetServerInformation](#), [GetServerStatus](#)

CFtpClient::GetServerType Method

```
INT GetServerType();
```

The **GetServerType** method returns the type of server the client has connected to.

Parameters

None.

Return Value

If the method succeeds, the return value is a numeric value which indicates the server type. If the method fails, the return value is `FTP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

This method sends the `SYST` command to the server to determine the server type. The following server types are recognized by the library:

Constant	Description
<code>FTP_SERVER_UNKNOWN</code>	The server type could not be determined by issuing the <code>SYST</code> command. The server may not support the command, or the command may only be allowed when issued by an authenticated user.
<code>FTP_SERVER_MSDOS</code>	The server is running on an MS-DOS based operating system. The server expects file pathing and naming conventions according to the standard MS-DOS format and returns directory listings similar to the output of the <code>DIR</code> command.
<code>FTP_SERVER_WINDOWS</code>	The server is running on a Windows based operating system. The server expects file pathing and naming conventions according to the standard Windows long filename format, and returns directory listings similar to the output of the <code>DIR</code> command. Note that Windows servers may be configured to return file and directory information in a format similar to UNIX systems, in which case the system may be identified as UNIX even though it is actually running on a Windows platform.
<code>FTP_SERVER_VMS</code>	The server is running on a DEC VMS based operating system. The server expects file pathing and naming conventions specific to that operating system. Note that VMS servers may be configured to return file and directory information in a format similar to UNIX systems, in which case the system may be identified as UNIX even though it is actually running on a VMS platform.
<code>FTP_SERVER_NETWARE</code>	The server is running on a NetWare based operating system. The server expects file pathing and naming conventions similar to the standard Windows long filename format, and returns directory listings that are similar to UNIX systems with the exception of the access and permissions flags for the file. Note that a NetWare system may return

	listings in different formats based on the filesystem and site specific options specified.
FTP_SERVER_OTHER	The server type was not recognized. An attempt will be made to automatically determine the correct file pathing and naming conventions used by the server. To obtain a list of files on the server, it may be necessary to use the SetDirectoryFormat method to specify the directory listing format.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[GetClientQuota](#), [GetServerInformation](#), [GetServerStatus](#), [SetDirectoryFormat](#)

CFtpClient::GetStatus Method

```
INT GetStatus();
```

The **GetStatus** method the current status of the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the client status code. If the method fails, the return value is `FTP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **GetStatus** method returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Constant	Description
0	<code>FTP_STATUS_UNUSED</code>	No connection has been established.
1	<code>FTP_STATUS_IDLE</code>	The client is current idle and not sending or receiving data.
2	<code>FTP_STATUS_CONNECT</code>	The client is establishing a connection with the server.
3	<code>FTP_STATUS_READ</code>	The client is reading data from the server.
4	<code>FTP_STATUS_WRITE</code>	The client is writing data to the server.
5	<code>FTP_STATUS_DISCONNECT</code>	The client is disconnecting from the server.
6	<code>FTP_STATUS_OPENFILE</code>	The client is opening a data connection to the server.
7	<code>FTP_STATUS_CLOSEFILE</code>	The client is closing the data connection to the server.
8	<code>FTP_STATUS_GETFILE</code>	The client is downloading a file from the server.
9	<code>FTP_STATUS_PUTFILE</code>	The client is uploading a file to the server.
10	<code>FTP_STATUS_FILELIST</code>	The client is retrieving a file listing from the server.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session. To obtain status information about a file transfer, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

See Also

[IsBlocking](#), [IsReadable](#), [IsWritable](#), [GetTransferStatus](#)

CFtpClient::GetText Method

```
INT GetText(  
    LPCTSTR lpszRemoteFile,  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);
```

The **GetText** method copies the contents of a text file on the server to the specified string buffer.

Parameters

lpszRemoteFile

A pointer to a string that specifies a text file on the server. The file pathing and naming conventions must be that of the host operating system.

lpszBuffer

A pointer to a string buffer which will contain the contents of the text file when the method returns. This buffer should be large enough to store the contents of the file, including a terminating null character. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. This value must be larger than zero. If this value is smaller than the actual size of the text file, the data returned will be truncated.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is `FTP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **GetText** method is used to download a text file and store the contents in a string buffer. This method will always set the file type to `FILE_TYPE_ASCII` before downloading the file, and will restore the default file type before the method returns. Because binary files can include embedded null characters, this method should only be used with known text files.

This method has been included as a convenience for applications that need to retrieve relatively small text files and manipulate the contents as a string. If the Unicode version of this method is called, the contents of the text file is automatically converted to a Unicode string. If the size of the file is unknown or the text file is very large, it is recommended that you use the **GetData** or **GetFile** methods.

If you use the **GetFileSize** method to determine how large the string buffer should be prior to calling this method, it is important to be aware that the actual number of characters may differ based on the end-of-line conventions used by the host operating system. For example, if you call **GetFileSize** to obtain the size of a text file on a UNIX system, the value will not be large enough to store the complete file because UNIX uses a single linefeed (LF) character to indicate the end-of-line, while a Windows system will use a carriage-return and linefeed (CRLF) pair. To accommodate this difference, you should always allocate extra memory for the string buffer to store the additional end-of-line characters.

`FTP_EVENT_PROGRESS` event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Example

```
LPTSTR lpszBuffer = (LPTSTR)calloc(MAXFILESIZE, sizeof(TCHAR));

if (lpszBuffer == NULL)
    return;

nResult = pClient->GetText(lpszRemoteFile, lpszBuffer, MAXFILESIZE);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ChangeDirectory](#), [EnableEvents](#), [GetData](#), [GetFile](#), [GetTransferStatus](#), [PutData](#), [PutFile](#), [RegisterEvent](#), [SetBufferSize](#)

CFtpClient::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[Connect](#), [IsReadable](#), [IsWritable](#), [Read](#), [SetTimeout](#), [Write](#)

CFtpClient::GetTransferStatus Method

```
INT GetTransferStatus(  
    LPFTPTRANSFERSTATUS lpStatus  
);  
  
INT GetTransferStatus(  
    LPFTPTRANSFERSTATUSEX lpStatus  
);
```

The **GetTransferStatus** method returns information about the current file transfer in progress.

Parameters

lpStatus

A pointer to an [FTPTRANSFERSTATUS](#) or [FTPTRANSFERSTATUSEX](#) structure which contains information about the status of the current file transfer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `FTP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **GetTransferStatus** method returns information about the current file transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no file currently being transferred, this method will return the status of the last successful transfer made by the client.

The *dwBytesTotal* and *dwBytesCopied* members of the **FTPTRANSFERSTATUSEX** structure are declared as unsigned 64-bit integers rather than 32-bit integers. To obtain accurate file transfer information, this extended version of the structure should be used with files that are larger than 4GiB.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session.

If the option `FTP_OPTION_HIRES_TIMER` has been specified when connecting to the server, the values of the *dwTimeElapsed* and *dwTimeEstimated* members of the **FTPTRANSFERSTATUS** and **FTPTRANSFERSTATUSEX** structure will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small files over a fast network connection.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableEvents](#), [GetFileStatus](#), [GetStatus](#), [RegisterEvent](#)

CFtpClient::IsBlocking Method

BOOL IsBlocking();

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking method may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Cancel](#), [GetStatus](#), [IsConnected](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#), [Read](#), [Write](#)

CFtpClient::IsConnected Method

```
BOOL IsConnected();
```

The **IsConnected** method is used to determine if the client is currently connected to a server.

Parameters

None.

Return Value

If the client is connected to a server, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#)

CFtpClient::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[CFtpClient](#), [IsBlocking](#), [IsConnected](#)

CFtpClient::IsReadable Method

```
BOOL IsReadable(  
    INT nTimeout,  
    LPDWORD lpdwAvail  
);
```

The **IsReadable** method is used to determine if data is available to be read from the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the function returns a non-zero value. If the client cannot read any data, the method returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsConnected](#), [IsInitialized](#), [IsWritable](#), [Write](#)

CFtpClient::IsWritable Method

```
BOOL IsWritable(  
    INT nTimeout  
);
```

The **IsWritable** method is used to determine if data can be written to the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the method returns a non-zero value. If the client cannot write any data, the method returns zero.

Remarks

Although this method can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: csftpv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsConnected](#), [IsInitialized](#), [IsReadable](#), [Write](#)

CFtpClient::Login Method

```
INT Login(  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszAccount  
);
```

The **Login** method authenticates the specified user in on the server. This method must be called after the connection has been established, and before attempting to transfer files or perform any other method on the server.

Parameters

lpszUserName

Points to a string that specifies the user name to be used to authenticate the current client session. If this parameter is NULL or an empty string, then the login is considered to be anonymous.

lpszPassword

Points to a string that specifies the password to be used to authenticate the current client session. This parameter may be NULL or an empty string if no password is required for the specified user, or if no username has been specified.

lpszAccount

Points to a string that specifies the account name to be used to authenticate the current client session. This parameter may be NULL or an empty string if no account name is required for the specified user.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

Some public FTP servers support anonymous logins, where a username and password are not required to access the server. In this case, both the *lpszUserName* and *lpszPassword* parameters can be NULL or specify empty strings. In most cases, access to the server using an anonymous login is restricted, with clients only having permission to download files. Servers may also restrict the maximum number of anonymous sessions that may be logged in at one time.

This method should only be used after calling the **Logout** method, enabling you to log in as another user during the same session. Not all servers will permit a client to change user credentials during the same session. In most cases, it is preferable to disconnect from the server and re-connect using the new credentials rather than using this method.

This method is not supported with secure connections using the SSH protocol.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [Logout](#), [ProxyConnect](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CFtpClient::Logout Method

INT Logout();

The **Logout** method logs out the user associated with the current client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is `FTP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **Logout** method is used when the client wants to re-authenticate using a new username and password. Before any further action may be taken, other than disconnecting from the server, the **Login** method must be called to re-authenticate the client.

It is not necessary to call this method prior to disconnecting from the server because the user current user is automatically logged out when the **Disconnect** method is called.

This method is not supported with secure connections using the SSH protocol.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [Disconnect](#), [Login](#)

CFtpClient::MountStructure Method

```
INT MountStructure(  
    LPCTSTR lpszFileSystem  
);
```

The **MountStructure** method mounts a different file system or other directory data structure on the server.

Parameters

lpszFileSystem

A pointer to a string which specifies the file system to mount on the server.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method sends the SMNT command to the server, which may not be supported on some platforms. Use of this command typically requires that the user have administrator privileges on the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateDirectory](#), [RemoveDirectory](#)

CFtpClient::OpenDirectory Method

```
INT OpenDirectory(  
    LPCTSTR lpszDirectory  
);
```

The **OpenDirectory** method opens the specified directory on the server.

Parameters

lpszDirectory

Pointer to the name of the directory that will be opened. The format of the directory name must match the filename conventions used by the server. If a NULL pointer or an empty string is specified, then the current working directory is opened.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **OpenDirectory** method opens the specified directory on the server using the LIST command. The contents of the directory can be read using the **GetFirstFile** and **GetNextFile** methods. The directory listing is returned on the data channel in one of several different formats. The library can recognize listing formats generated by UNIX, VMS and Windows servers, as well as those of other servers which emulate one of those common formats. Once the complete directory listing has been read, the directory must be closed by calling the **CloseDirectory** method.

Because the directory listing is returned on the data channel, a file transfer cannot be performed while the directory is in the process of being read by the client. Applications which need to collect a list of files to download should first open the directory, read the contents and store the file names in an array. After the directory has been closed, the application can then start transferring the files to the local system.

Some servers may not support file listings for any directory other than the current working directory. If an error is returned when specifying a directory name, try changing the current working directory using the **ChangeDirectory** method and then call this method again, passing NULL or an empty string as the *lpszDirectory* parameter.

To obtain a list of all files in a directory using a single function call, use the **EnumFiles** method. If the server lists files in a format that is not recognized by the library, the **GetFileList** method can be used to obtain an unparsed file listing from the server.

Example

```
if (pClient->OpenDirectory() != FTP_ERROR)  
{  
    FTPFILESTATUS ftpFile;  
    BOOL bResult;  
  
    bResult = pClient->GetFirstFile(&ftpFile);  
    while (bResult)  
    {  
        // The ftpFile structure contains information about the file  
        bResult = pClient->GetNextFile(&ftpFile);  
    }  
}
```

```
pClient->CloseDirectory();  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ChangeDirectory](#), [CloseDirectory](#), [EnumFiles](#), [GetDirectoryFormat](#), [GetFileList](#), [GetFileStatus](#), [GetFirstFile](#), [GetNextFile](#), [SetDirectoryFormat](#)

CFtpClient::OpenFile Method

```
INT OpenFile(  
    LPCTSTR lpszFileName,  
    DWORD dwOpenMode,  
    DWORD dwOffset  
);
```

The **OpenFile** method creates or opens the specified file on the server.

Parameters

lpszFileName

Points to a string that specifies the name of the remote file to create or open. The file pathing and name conventions must be that of the server.

dwOpenMode

Specifies the type of access to the file. An application can open a file for reading, create a new file or append data to an existing file. This parameter should be one of the following values.

Constant	Description
FTP_FILE_READ	The file is opened for reading on the server. A data channel is created and the contents of the file are returned to the client.
FTP_FILE_WRITE	The file is opened for writing on the server. If the file does not exist, it will be created. If it does exist, it will be overwritten.
FTP_FILE_APPEND	The file is opened for writing on the server. All data will be appended to the end of the file.

dwOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset using FTP requires that the server support the REST command to restart transfers.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

Only one file may be opened at a time for each client session. Attempting to perform an action such as uploading or downloading another file while a file is currently open will result in an error. Typically this indicates that the application failed to call the **CloseFile** method.

It is strongly recommended that most applications use the **GetFile** or **PutFile** methods to perform file transfers. These methods are easier to use, and have internal optimizations that improves the overall data transfer rate when compared to implementing the file transfer code in your own application.

When a file is created on the server, the file ownership and access rights are determined by the server. Some servers may provide a method to change these attributes through site-specific commands. Refer to the server's operating system documentation for more information about

what commands may be available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CloseFile](#), [GetFile](#), [GetFileSize](#), [GetFileTime](#), [PutFile](#), [SetFileTime](#)

CFtpClient::ProxyConnect Method

```
HCLIENT ProxyConnect(  
    UINT nProxyType,  
    LPCTSTR LpszProxyHost,  
    UINT nProxyPort,  
    LPCTSTR LpszProxyUser,  
    LPCTSTR LpszProxyPassword,  
    LPCTSTR LpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **ProxyConnect** method establishes a connection through a proxy server.

Parameters

nProxyType

An identifier which specifies the type of proxy server that is being connected to. This value must be defined as one of the following values:

Constant	Description
FTP_PROXY_NONE	This value specifies that no proxy server is being used. In this case, the Connect method is called directly, ignoring the proxy parameters.
FTP_PROXY_USER	This value specifies that the client is not logged into the proxy server. The USER command is sent in the format username@ftpsite followed by the password. This is the format used with the Gauntlet proxy server.
FTP_PROXY_LOGIN	This value specifies that the client is logged into the proxy server. The USER command is then sent in the format username@ftpsite followed by the password. This is the format used by the InterLock proxy server.
FTP_PROXY_OPEN	This value specifies that the client is not logged into the proxy server. The OPEN command is sent specifying the host name, followed by the username and password.
FTP_PROXY_SITE	This value specifies that the client is logged into the server. The SITE command is sent, specifying the host name, followed by the username and the password.
FTP_PROXY_OTHER	This special proxy type specifies that another, undefined proxy server is being used. The client connects to the proxy host, but does not attempt to authenticate the client. The application is responsible for negotiating with the proxy server, typically using the Command method to send specific command sequences.

lpszProxyHost

A pointer to the name of the proxy server to connect through; this may be a fully-qualified

domain name or an IP address.

lpszProxyPort

The port number the proxy server is listening on; a value of zero specifies that the default port number should be used.

lpszProxyUser

A pointer to the user name used to authenticate the client on the proxy server. Not all proxy servers require this information; it is recommended that you consult the proxy server documentation to determine if a username is required.

lpszProxyPassword

A pointer to the password used to authenticate the client on the proxy server. Not all proxy servers require this information; it is recommended that you consult the proxy server documentation to determine if a password is required.

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used. For standard connections, the default port number is 21. For secure connections, the default port number is 990.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
FTP_OPTION_PASSIVE	This option specifies the client should attempt to establish the data connection with the server. When the client uploads or downloads a file, normally the server establishes a second connection back to the client which is used to transfer the file data. However, if the local system is behind a firewall or a NAT router, the server may not be able to create the data connection and the transfer will fail. By specifying this option, it forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for

	the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_NOAUTH	This option specifies the server does not require authentication, or that it requires an alternate authentication method. When this option is used, the client connection is flagged as authenticated as soon as the connection to the server has been established. Note that using this option to bypass authentication may result in subsequent errors when attempting to retrieve a directory listing or transfer a file. It is recommended that you consult the technical reference documentation for the server to determine its specific authentication requirements.
FTP_OPTION_VIRTUALHOST	This option specifies the server supports virtual hosting, where multiple domains are hosted by a server using the same external IP address. If this option is enabled, the client will send the HOST command to the server upon establishing a connection.
FTP_OPTION_VERIFY	This option specifies that file transfers should be automatically verified after the transfer has completed. If the server supports the XMD5 command, the transfer will be verified by calculating an MD5 hash of the file contents. If the server does not support the XMD5 command, but does support the XCRC command, the transfer will be verified by calculating a CRC32 checksum of the file contents. If neither the XMD5 or XCRC commands are supported, the transfer is verified by comparing the size of the file. Automatic file verification is only performed for binary mode transfers because of the end-of-line conversion that may occur when text files are uploaded or downloaded.
FTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. This option is the same as specifying FTP_OPTION_SECURE_IMPLICIT which immediately performs the SSL/TLS protocol negotiation when the connection is established.
FTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to immediately establish secure SSL/TLS connection with the server. This option is typically used when

	connecting to a server on port 990, which is the default port number used for FTPS.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should establish a standard connection to the server and then use the AUTH command to negotiate an explicit secure connection. This option is typically used when connecting to the server on ports other than 990.
FTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
FTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established. This option also forces all connections to be outbound and enables the firewall compatibility features in the client.
FTP_OPTION_KEEPALIVE_DATA	This option specifies the client should attempt to keep the control connection active during a file transfer. Normally, when a data transfer is in progress, no additional commands are issued on the control channel until the transfer completes. Specifying this option automatically enables the FTP_OPTION_KEEPALIVE option and forces the client to continue to issue NOOP commands during the file transfer. This option only applies to FTP and FTPS connections and has no effect on connections using SFTP (SSH).
FTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
FTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
FTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data

	transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller files being uploaded or downloaded using fast network connections.
FTP_OPTION_TLS_REUSE	This option specifies that TLS session reuse should be enabled for secure connections. This option is only supported on Windows 8.1 or Windows Server 2012 R2 and later platforms, and it should only be used when explicitly required by the server. This option is not compatible with servers built using OpenSSL 1.0.2 and earlier versions which do not provide Extended Master Secret (EMS) support as outlined in RFC7627.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur. If this argument is NULL, then the client session will be blocking and no network events will be sent to the client.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the *hEventWnd* argument is NULL, this argument should be specified as WM_NULL.

Return Value

If the method succeeds, the return value is a handle to a client session. If the method fails, the return value is INVALID_CLIENT. To get extended error information, call **GetLastError**.

Remarks

The use of Windows event notification messages has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

To prevent this method from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **Connect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

The username and password that is used to authenticate the client with the proxy server are not the same as those used to login to the target server. Once a connection has been established with the proxy server, the client must call the **Login** method to actually login to the server and begin a file transfer.

If you specify an event notification window, then the client session will be asynchronous. When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description

FTP_EVENT_CONNECT	The control connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
FTP_EVENT_DISCONNECT	The server has closed the control connection to the client. The client should read any remaining data and disconnect.
FTP_EVENT_OPENFILE	The data connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
FTP_EVENT_CLOSEFILE	The server has closed the data connection to the client. The client should read any remaining data and close the data channel.
FTP_EVENT_READFILE	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_WRITEFILE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
FTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and reconnect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
FTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
FTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.
FTP_EVENT_GETFILE	This event is generated when a file download has completed. If multiple files are being downloaded, this event will be generated for each file.
FTP_EVENT_PUTFILE	This event is generated when a file upload has completed. If multiple files are being uploaded, this event will be generated for each file.

To cancel asynchronous notification and return the client to a blocking mode, use the **DisableEvents** method.

It is recommended that you only establish an asynchronous connection if you understand the

implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

If the `FTP_OPTION_KEEPALIVE` option is specified, a background worker thread will be created to monitor the command channel and periodically send NOOP commands to the server if no commands have been sent recently. This can prevent the server from terminating the client connection during idle periods where no commands are being issued. However, it is important to keep in mind that many servers can be configured to also limit the total amount of time a client can be connected to the server, as well as the amount of time permitted between file transfers. If the server does not respond to the NOOP command, this option will be automatically disabled for the remainder of the client session.

If the `FTP_OPTION_SECURE_EXPLICIT` option is specified, the client will first send an AUTH TLS command to the server. If the server does not accept this command, it will then send an AUTH SSL command. If both commands are rejected by the server, an explicit SSL session cannot be established. By default, both the command and data channels will be encrypted when a secure connection is established. To change this, use the **SetChannelMode** method.

The *dwOptions* argument can be used to specify the threading model that is used by the class when a connection is established. By default, the client session is initially attached to the thread that created it. From that point on, until the connection is terminated, only the owner may invoke methods in that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the `FTP_OPTION_FREETHREAD` option enables any thread to call methods in any instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the class and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a method in the class, it must ensure that no other thread will call another method at the same time using the same instance.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [CreateSecurityCredentials](#), [Disconnect](#), [GetSecurityInformation](#), [Login](#)

CFtpClient::PutData Method

```
INT PutData(  
    LPCTSTR lpszRemoteFile,  
    LPBYTE lpBuffer,  
    DWORD dwLength  
);
```

```
INT PutData(  
    LPCTSTR lpszRemoteFile,  
    LPCTSTR lpszBuffer  
);
```

The **PutData** method transfers the contents of the specified buffer to a file on the server.

Parameters

lpszRemoteFile

A pointer to a string that specifies the file on the server that will be created, overwritten or appended to. The file naming conventions must be that of the host operating system.

lpBuffer

A pointer to the data that will be copied to the server and stored in the specified file. An alternate version of the method uses a pointer to a string buffer where all of the bytes will be written to the server up to, but not including, the terminating null character.

dwLength

The number of bytes to copy from the buffer.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is `FTP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the `FTP_EVENT_PROGRESS` event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ChangeDirectory](#), [EnableEvents](#), [GetData](#), [GetFile](#), [GetTransferStatus](#), [PutFile](#), [RegisterEvent](#), [SetBufferSize](#)

CFtpClient::PutFile Method

```
INT PutFile(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszRemoteFile,  
    DWORD dwOptions  
    DWORD dwOffset  
);
```

The **PutFile** method transfers the specified file on the local system to the server.

Parameters

lpszLocalFile

A pointer to a string that specifies the file that will be transferred from the local system. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A pointer to a string that specifies the file on the server that will be created, overwritten or appended to. The file naming conventions must be that of the host operating system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Constant	Description
FTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the remote file exists, it will be overwritten with the contents of the uploaded file.
FTP_TRANSFER_APPEND	This option specifies that if the remote file exists, the contents of the local file is appended to the remote file. If the remote file does not exist, it is created.

dwOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ChangeDirectory](#), [EnableEvents](#), [GetData](#), [GetFile](#), [GetMultipleFiles](#), [GetTransferStatus](#), [PutData](#), [PutMultipleFiles](#), [RegisterEvent](#), [SetBufferSize](#), [VerifyFile](#)

CFtpClient::PutMultipleFiles Method

```
INT PutMultipleFiles(  
    LPCTSTR lpszLocalDirectory,  
    LPCTSTR lpszRemoteDirectory,  
    LPCTSTR lpszFileMask  
);
```

The **PutMultipleFiles** method copies one or more files from the local host to the server, using the specified wildcard.

Parameters

lpszLocalDirectory

Pointer to a string which specifies the local directory where the files will be copied from. A NULL pointer or empty string specifies that files should be copied from the current working directory.

lpszRemoteDirectory

Pointer to a string which specifies the remote directory where the files will be copied to. A NULL pointer or empty string specifies that the files should be copied to the current working directory on the server.

lpszFileMask

Pointer to a string which specifies the files that are to be copied from the local system to the server. The file mask should follow the Windows conventions used for wildcard file matches.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **PutMultipleFiles** method is used to transfer files from the local host to the server which match a specified wildcard file mask. All files will be transferred using the current file type as specified by the **SetFileType** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ChangeDirectory](#), [GetFile](#), [GetMultipleFiles](#), [PutFile](#)

CFtpClient::PutText Method

```
INT PutText(  
    LPCTSTR lpszRemoteFile,  
    LPCTSTR lpszBuffer  
);
```

The **PutText** method creates a text file on the server using the contents of a string buffer.

Parameters

lpszRemoteFile

A pointer to a string that specifies the text file on the server that will be created or overwritten. The file pathing and name conventions must be that of the server.

lpszBuffer

A pointer to a string that contains the text that will be stored in the file.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **PutText** method is used to create a text file on the server from the contents of a string. If the specified file already exists on the server, its contents will be overwritten. This method will always set the file type to FILE_TYPE_ASCII before creating the file, and will restore the default file type before the method returns.

If the Unicode version of this method is called, the string will be converted to ASCII and then uploaded to the server. If you wish to store the contents of the string as Unicode on the server, you must set the current file type to FILE_TYPE_IMAGE and use the **PutData** method. This method should never be used to upload binary data.

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ChangeDirectory](#), [EnableEvents](#), [GetText](#), [GetTransferStatus](#), [PutData](#), [PutFile](#), [RegisterEvent](#), [SetBufferSize](#)

CFtpClient::Read Method

```
INT Read(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Read(  
    CString& strBuffer,  
    INT cbBuffer  
);
```

The **Read** method reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be copied. An alternate form of the method accepts a **CString** object which will contain the data returned from the server.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the method fails, the return value is `FTP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

When **Read** is called and the client is in non-blocking mode, it is possible that the method will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableEvents](#), [GetData](#), [GetFile](#), [IsBlocking](#), [IsReadable](#), [IsWritable](#), [RegisterEvent](#), [Write](#)

CFtpClient::RegisterEvent Method

```
INT RegisterEvent(  
    UINT nEventId,  
    FTPEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Constant	Description
FTP_EVENT_CONNECT	The control connection to the server has completed.
FTP_EVENT_DISCONNECT	The server has closed the control connection to the client. The client should read any remaining data and disconnect.
FTP_EVENT_OPENFILE	The data connection to the server has completed.
FTP_EVENT_CLOSEFILE	The server has closed the data connection to the client. The client should read any remaining data and close the data channel.
FTP_EVENT_READFILE	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_WRITEFILE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
FTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and reconnect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
FTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
FTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a

	status control or progress bar.
FTP_EVENT_GETFILE	This event is generated when a file download has completed. If multiple files are being downloaded, this event will be generated for each file.
FTP_EVENT_PUTFILE	This event is generated when a file upload has completed. If multiple files are being uploaded, this event will be generated for each file.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **FtpEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This method is typically used to register an event handler that is invoked while a file is being uploaded or downloaded. The FTP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one FTP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

CFtpClient::RegisterFileType Method

```
BOOL RegisterFileType(  
    LPCTSTR lpszExtension,  
    UINT nFileType  
);
```

The **RegisterFileType** method associates a file name extension with a specific file type.

Parameters

lpszExtension

A pointer to a null terminated string which specifies the file name extension. If this parameter is NULL or points to an empty string, the default file type will be changed for the client session.

nFileType

Specifies the type of file associated with the file extension. This parameter can be one of the following values.

Value	Description
FILE_TYPE_ASCII	The file is a text file using the ASCII character set. For those servers which mark the end of a line with characters other than a carriage return and linefeed, it will be converted to the native client format. This is the file type used for directory listings.
FILE_TYPE_EBCDIC	The file is a text file using the EBCDIC character set. Local files will be converted to EBCDIC when sent to the server. Remote files will be converted to the native ASCII character set when retrieved from the server.
FILE_TYPE_IMAGE	The file is a binary image and no data conversion of any type is performed on the file. This is typically the default file type for data file transfers. If the type of file that is being transferred is unknown, this file type should always be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **RegisterFileType** method is used to associate specific file types with file name extensions. The library has an internal list of standard text file extensions which it automatically recognizes. This method can be used to extend or modify that list for the client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[OpenFile](#), [SetFileMode](#), [SetFileStructure](#), [SetFileType](#), [SetPassiveMode](#)

CFtpClient::RemoveDirectory Method

```
INT RemoveDirectory(  
    LPCTSTR lpszDirectory  
);
```

The **RemoveDirectory** method removes the specified directory on the server.

Parameters

lpszDirectory

Points to a string that specifies the name of the directory. The file pathing and name conventions must be that of the server.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method uses the RMD command to create the directory. The user must have the appropriate permission to remove the specified directory. Most servers will not permit you to remove a directory if it contains one or more files.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ChangeDirectory](#), [CreateDirectory](#), [DeleteFile](#), [GetDirectory](#)

CFtpClient::RenameFile Method

```
INT RenameFile(  
    LPCTSTR lpszOldFileName,  
    LPCTSTR lpszNewFileName  
);
```

The **RenameFile** method renames the specified file on the server. The file must exist, and the current user must have the appropriate permission to change the file name.

Parameters

lpszOldFileName

Points to a string that specifies the name of the remote file to rename. The file pathing and name conventions must be that of the server.

lpszNewFileName

Points to a string the specifies the new name for the remote file. The file pathing and name conventions must be that of the server.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method causes two separate commands to be sent to the server, RNFR and RNT0. If either command fails, the method will fail and return an error code.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DeleteFile](#), [GetFile](#), [PutFile](#)

CFtpClient::Reset Method

```
INT Reset();
```

The **Reset** method resets the client state and resynchronizes with the server. This method is typically called after an unexpected error has occurred, or an operation has been canceled.

Parameters

None.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The client cannot be reset while a file transfer is in progress or if the client is in a blocked state. To abort a file transfer, use the **Cancel** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Cancel](#)

CFtpClient::SetActivePorts Method

```
INT SetActivePorts(  
    UINT nLowPort,  
    UINT nHighPort  
);
```

The **SetActivePorts** method changes the range of local port numbers used for active mode file transfers.

Parameters

hClient

Handle to the client session.

nLowPort

An unsigned integer that specifies the low port number.

lpnHighPort

An unsigned integer that specifies the high port number.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This method is used to modify the range of local port numbers used for active mode file transfers. When using active mode, the client listens for an inbound connection from the server rather than establishing an outbound connection for the data transfer. In most cases, passive mode transfers are preferred because they mitigate potential compatibility issues with firewalls and NAT routers.

If active mode transfers are required, the default port range used when listening for the server connection is between 1024 and 5000. This is the standard range of ephemeral ports used by the Windows operating system. However, under some circumstances that range of ports may be too small, or a firewall may be configured to deny inbound connections on ephemeral ports. In that case, the **SetActivePorts** method can be used to specify a different range of port numbers.

While it is technically permissible to assign the low and high port numbers to the same value, effectively specifying a single active port number, this is not recommended as it can cause the transfer to fail unexpectedly if multiple file transfers are performed. A minimum range of at least 1000 ports is recommended. For example, if you specify a low port value of 40000 then it is recommended that the high port value be at least 41000. The maximum port value is 65535.

To determine the current range of active port numbers being used, call the **GetActivePorts** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetActivePorts](#), [SetPassiveMode](#)

CFtpClient::SetBufferSize Method

```
INT SetBufferSize(  
    INT nBufferSize  
);
```

The **SetBufferSize** method sets the size in bytes of an internal buffer that will be used during data transfers.

Parameters

nBufferSize

The size of an internal buffer, in bytes. Any value greater than or equal to zero is acceptable. If *nBufferSize* is zero, then the default value of 4096 will be used. If *nBufferSize* is less than 256 bytes, the buffer size will be set to 256. The maximum value of *nBufferSize* is 1048576 (1Mb).

Return Value

If the method succeeds, the return value is the size of the internal buffer that will be used. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The speed of data transfers, particularly on uploads, may be sensitive to network type and configuration, and the size of the internal buffer used for data transfers. The default size of this buffer will result in good performance for a wide range of network characteristics. A larger buffer will not necessarily result in better performance. For example, a multiple of 1460, which is the typical Maximum Transmission Unit (MTU), may be optimal in many situations.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetBufferSize](#), [GetData](#), [GetFile](#), [PutData](#), [PutFile](#)

CFtpClient::SetChannelMode Method

```
INT SetChannelMode(  
    INT nMode  
);  
  
INT SetChannelMode(  
    INT nChannel,  
    INT nMode  
);
```

The **SetChannelMode** method changes the security mode for the specified communication channel.

Parameters

nChannel

An integer value which specifies which channel to return information for. If the first version of this method is called, then the data channel mode is modified; otherwise, the channel may be explicitly specified as one of the following values:

Constant	Description
FTP_CHANNEL_COMMAND	Change information for the command channel. This is the communication channel used to send commands to the server and receive command result and status information from the server.
FTP_CHANNEL_DATA	Change information for the data channel. This is the communication channel used to send or receive data during a file transfer.

nMode

An integer value which specifies the new mode for the specified channel. It may be one of the following values:

Constant	Description
FTP_CHANNEL_CLEAR	Data sent and received on this channel should not be encrypted.
FTP_CHANNEL_SECURE	Data sent and received on this channel should be encrypted. Specifying this option requires that a secure connection has already been established with the server.

Return Value

If the method succeeds, the return value is the previous mode for the specified channel. If the method fails, it will return `FTP_ERROR`. To get extended error information, call **FtpGetLastError**.

Remarks

The **SetChannelMode** method is used to change the default mode for the specified channel, and is typically used to control whether or not data is encrypted during a file transfer. If a standard, non-secure connection has been established with the server, an error will be returned if you specify the `FTP_CHANNEL_SECURE` mode for either channel.

If you have established a secure connection and then specify the `FTP_CHANNEL_CLEAR` mode for

the command channel, the client will send the CCC command to the server to indicate that commands should no longer be encrypted. If the server does not support this command, an error will be returned and the channel mode will remain unchanged. Once the command channel has been changed to clear mode, it cannot be changed back to secure mode. You must disconnect and re-connect to the server if you want to resume sending commands over an encrypted channel.

Changing the mode for the data channel requires that the server support the PROT command. If this command is not supported by the server, the method will fail and the channel mode will remain unchanged.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[GetChannelMode](#)

CFtpClient::SetDirectoryFormat Method

```
INT SetDirectoryFormat(  
    INT nFormatId  
);
```

The **SetDirectoryFormat** method is used to specify the format used by the server when returning a list of files. The format type is used internally by the library when parsing the file list returned by the server.

Parameters

nFormatId

An identifier used to specify the format of the file list returned by the server. The following values are recognized:

Constant	Description
FTP_DIRECTORY_AUTO	This value specifies that the library should automatically determine the format of the file lists returned by the server. It is recommended that most applications use this value and allow the library to automatically determine the appropriate file listing format used by the server.
FTP_DIRECTORY_UNIX	This value specifies that the server returns file lists in the format commonly used by UNIX servers. Note that many servers can be configured to return file listings in this format, even if they are not actually a UNIX based platform. Consult the technical reference documentation for your server for more information.
FTP_DIRECTORY_MSDOS	This value specifies that the server returns file lists in the format commonly used by MS-DOS based systems. This includes Windows IIS servers. Long file names will be returned if supported by the underlying filesystem, such as NTFS or FAT32.
FTP_DIRECTORY_VMS	This value specifies that the server returns file lists in the format commonly used by VMS servers. Note that VMS servers can be configured to return a standard UNIX style listing in addition to the default VMS format.
FTP_DIRECTORY_STERLING_1	This value specifies that the server returns file listings in a proprietary format used by the Sterling server, which is used for EDI (Electronic Data Interchange) applications. This format uses a 13 byte status code.
FTP_DIRECTORY_STERLING_2	This value specifies that the server returns file listings in a proprietary format used by the Sterling server, which is used for EDI (Electronic Data Interchange) applications. This format uses a 10 byte status code.
FTP_DIRECTORY_NETWARE	This value specifies that the server returns file listings in a proprietary format used by NetWare servers. The

	format is similar to UNIX style listings except that file access and permissions are indicated by letter codes enclosed in brackets. This is the default format selected if the server identifies itself as a NetWare system.
FTP_DIRECTORY_MLSD	This value specifies that the server should return file listings in a machine-independent format as defined by RFC 3659. This format specifies file information as a sequence of name/value pairs, with the same format being used regardless of the operating system that the server is hosted on. Note that not all servers support this format, and some proxy servers may reject the command even if the remote server supports its use.

Return Value

If the method succeeds, the return value identifies the file list format used by the server. If the method fails, the return value is `FTP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

This method should only be used when the library cannot automatically determine the directory format returned by the server. To determine the format used by a server after a file list has been retrieved, use the **GetDirectoryFormat** method.

The default directory format is determined both by the server's operating system and by analyzing the format of the data returned by the server. If the library is unable to automatically determine the format, it will attempt to parse the list of files as though it is a UNIX style listing.

If the `FTP_DIRECTORY_MLSD` format is specified, the file information returned by the server may differ from the default output of the `LIST` command. For example, on a UNIX based FTP server, the output of the `LIST` command is typically the same format that is used by the `/bin/lis` command, where file names are sorted and hidden files are not listed. However, the `MLSD` command may return an unsorted list of files that includes hidden files and directories.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CloseDirectory](#), [GetDirectoryFormat](#), [GetFileStatus](#), [GetFirstFile](#), [GetNextFile](#), [OpenDirectory](#)

CFtpClient::SetFeatures Method

```
DWORD SetFeatures(  
    DWORD dwFeatures  
);
```

The **SetFeatures** method specifies the server features available to the client.

Parameters

dwFeatures

An unsigned integer that specifies one or more features. Refer to the documentation for the **GetFeatures** method for a list of available features.

Return Value

If the method succeeds, the return value specifies the features that were previously enabled. If the method fails, it will return zero. Because it is possible that no features were enabled, a return value of zero does not always indicate an error. An application should call **GetLastError** to determine if an error code has been set.

Remarks

The **SetFeatures** method is used to enable a specific set of features for the current session. When a client connection is first established, all features are enabled based on the server type and the server's response to the FEAT command. However, as the client issues commands to the server, if the server reports that the command is unrecognized that feature will automatically be disabled in the client. To enable or disable a specific feature, an application can use the **EnableFeature** method.

For example, the first time an application calls the **GetFileSize** method to determine the size of a file, the library will try to use the SIZE command. If the server reports that the SIZE command is not available, that feature will be disabled and the library will not use the command again during the session unless it is explicitly re-enabled. This is designed to prevent the library from repeatedly sending invalid commands to a server, which may result in the server aborting the connection.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

See Also

[EnableFeature](#), [GetFeatures](#)

CFtpClient::SetFileMode Method

```
INT SetFileMode(  
    UINT nMode  
);
```

The **SetFileMode** method sets the default file transfer mode for the current client session.

Parameters

nMode

Specifies the default type of data transfer mode for files being opened or created on the server. This parameter can be one of the following values.

Value	Description
FILE_MODE_STREAM	The data is transmitted as a stream of bytes. This is the default client transfer mode.
FILE_MODE_BLOCK	The data is transmitted as a series of data blocks preceded by one or more header bytes. This transfer mode is currently not supported.
FILE_MODE_COMPRESSED	The data is transmitted in compressed form. This transfer mode is currently not supported.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The file transfer mode should be set before a file is opened or created on the server. Once the transfer mode is set, it is in effect for all files that are subsequently opened or created.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[OpenFile](#), [SetFileStructure](#), [SetFileType](#), [SetPassiveMode](#)

CFtpClient::SetFileNameEncoding Method

```
INT SetFileNameEncoding(  
    INT nEncoding  
);
```

The **SetFileNameEncoding** method specifies what type of encoding will be used when file names are sent to the server.

Parameters

nEncoding

An integer value which specifies the encoding type. It may be one of the following values:

Constant	Description
FTP_ENCODING_ANSI	File names are sent as 8-bit characters using the default character encoding for the current codepage. If the Unicode version of the functions are used, file names are converted from Unicode to ANSI using the current codepage before being sent to the server. This is the default encoding type.
FTP_ENCODING_UTF8	File names that contain non-ASCII characters are sent using UTF-8 encoding. This encoding type is only available on servers that advertise support for UTF-8 encoding and permit that encoding type to be enabled by the client.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **SetFileNameEncoding** method can be used to enable UTF-8 encoding of file names, which provides improved support for the use of international character sets. However, the server must provide support for UTF-8 encoding by advertising it in response to the FEAT command and it must support the OPTS command which is used to enable UTF-8 encoding. If the server does not advertise support for UTF-8, or the OPTS command fails with an error, then this method will fail with an error and the encoding type will not change.

Although it is possible to use the **EnableFeature** method to explicitly enable the FTP_FEATURE_UTF8 feature, this is not recommended. If the server has not advertised support for UTF-8 encoding in response to the FEAT command, that typically indicates that UTF-8 encoding is not supported. Attempting to force UTF-8 encoding can result in unpredictable behavior when file names contain non-ASCII characters.

It is important to note that not all FTP servers support UTF-8 encoding, and in some cases servers which advertise support for UTF-8 encoding do not implement the feature correctly. For example, a server may allow a client to enable UTF-8 encoding, but once enabled will not permit the client to disable it. Some servers may advertise support for UTF-8 encoding, however if the underlying file system does not support UTF-8 encoded file names, any attempt to upload or download a file may fail with an error indicating that the file cannot be found or created.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[Command](#), [EnableFeature](#) [GetFeatures](#), [GetFileNameEncoding](#), [SetFeatures](#)

CFtpClient::SetFilePermissions Method

```
INT SetFilePermissions(  
    LPCTSTR lpszFileName,  
    DWORD dwPermissions  
);
```

The **SetFilePermissions** method returns information about the access permissions for a specific file on the server.

Parameters

lpszFileName

A pointer to a string which contains the name of the file to be updated. The filename cannot contain any wildcard characters.

dwPermissions

An unsigned integer which will specify the new access permissions for the file. The file permissions are represented as bit flags, and may be one or more of the following values combined with a bitwise Or operator:

Constant	Description
FILE_OWNER_READ	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.
FILE_OWNER_WRITE	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_OWNER_EXECUTE	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_GROUP_READ	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
FILE_GROUP_WRITE	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_GROUP_EXECUTE	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_WORLD_READ	All users have permission to open the file for reading. This permission grants any user the right to download the file to

	the local system.
FILE_WORLD_WRITE	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.
FILE_WORLD_EXECUTE	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.

Return Value

If the method succeeds, the return value is a result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method uses the SITE CHMOD command to set the permissions for the file. This command is typically only supported on servers that are hosted on UNIX based systems. If the command is not supported, an error will be returned.

Users who are familiar with the UNIX operating system will recognize the **chmod** command used to change the file permissions. However, it should be noted that the numeric value used as an argument to the command is in octal, not decimal. For example, issuing the command **chmod 644 filename.txt** on a UNIX based system will make the file readable and writable by the owner, and readable by other users in the owner's group as well as all other users. The value 644 is an octal value, which is equivalent to the decimal value 420. If you were to mistakenly specify 644 as the value for the *dwPermissions* parameter, rather than the decimal value of 420, the permissions on the file would be incorrect. It is strongly recommended that you use the pre-defined constants to prevent this sort of error.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFilePermissions](#), [GetFileStatus](#)

CFtpClient::SetFileStructure Method

```
INT SetFileStructure(  
    UINT nType  
);
```

The **SetFileStructure** method sets the default file structure for the current client session, which indicates what type of file is being opened or created on the server.

Parameters

nType

Specifies the default type of file structure being opened or created on the server. This parameter can be one of the following values.

Value	Description
FILE_STRUCT_NONE	The file has no inherent structure and is considered to be a stream of bytes. This is the default structure for file transfers.
FILE_STRUCT_RECORD	The file uses a record structure. This file structure is currently not supported.
FILE_STRUCT_PAGE	The file uses a page structure. This file structure is currently not supported.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The file structure should be set before a file is opened or created on the server. Once the file type is set, it is in effect for all files that are subsequently opened or created.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[OpenFile](#), [SetFileMode](#), [SetFileType](#), [SetPassiveMode](#)

CFtpClient::SetFileTime Method

```
INT SetFileTime(  
    LPCTSTR lpszFileName,  
    LPSYSTEMTIME lpFileTime  
);
```

The **SetFileTime** method sets the modification time for the specified file on the server.

Parameters

lpszFileName

Points to a string that specifies the name of the remote file.

lpFileTime

Points to a [SYSTEMTIME](#) structure that specifies the new modification time for the remote file.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SetFileTime** method will change the modification time of a file on the server. The values specified in the SYSTEMTIME structure are expected to represent UTC time, not time adjusted for the local system's timezone. If the values do represent the local time, it must be converted to UTC time prior to calling this method. To populate the SYSTEMTIME structure with the current time, use the **GetSystemTime** method.

When connected to an FTP server, this method uses the MDTM command to set the modification time for the specified file. Not all servers implement this command, in which case the method call will fail. Note that some servers only support the MDTM command to return, but not change, the file modification time.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFileStatus](#), [GetFileTime](#), [OpenDirectory](#)

CFtpClient::SetFileType Method

```
INT SetFileType(  
    UINT nFileType  
);
```

The **SetFileType** method sets the default file type for the current client session, which indicates what type of file is being opened or created on the server.

Parameters

hClient

Handle to the client session.

nFileType

Specifies the default type of file being opened or created on the server. This parameter can be one of the following values.

Value	Description
FILE_TYPE_AUTO (0)	The file type should be automatically determined based on the file name extension. If the file extension is unknown, the file type should be determined based on the contents of the file. The library has an internal list of common text file extensions, and additional file extensions can be registered using the FtpRegisterFileType function.
FILE_TYPE_ASCII (1)	The file is a text file using the ASCII character set. For those servers which mark the end of a line with characters other than a carriage return and linefeed, it will be converted to the native client format. This is the file type used for directory listings.
FILE_TYPE_EBCDIC (2)	The file is a text file using the EBCDIC character set. Local files will be converted to EBCDIC when sent to the server. Remote files will be converted to the native ASCII character set when retrieved from the server. Not all servers support this file type. It is recommended that you only specify this type if you know that it is required by the server to transfer data correctly.
FILE_TYPE_IMAGE (3)	The file is a binary file and no data conversion of any type is performed on the file. This is the default file type for most data files and executable programs. If the type of file cannot be automatically determined, it will always be considered a binary file. If this file type is specified when uploading or downloading text files, the native end-of-line character sequences will be preserved.
FILE_TYPE_LOCAL (4)	The file is a binary file that uses the local byte size for the server platform. On most servers, this file type is considered to be the same as FILE_TYPE_IMAGE. Not all servers support this file type. It is recommended that you only specify this type if you know that it is required by the server to transfer data correctly.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The file type should be set before a file is opened or created on the server. Once the file type is set, it is in effect for all files that are subsequently opened or created. Some methods, such as **OpenDirectory** and **GetText**, will temporarily change the default file type to FILE_TYPE_ASCII and then restore the current file type when they return.

Calling this method has no practical effect when connected to an SFTP (SSH) server. They do not differentiate between text and binary files and the default file type will always be FILE_TYPE_IMAGE. If your application is uploading or downloading a text file, this difference between FTP and SFTP is important because the operating system that hosts the server may have different end-of-line character conventions than the client system. For example, if you download a text file from a UNIX system using SFTP, the end-of-line is indicated by a single linefeed (LF) character. However, on the Windows platform, the end-of-line is indicated by a carriage-return and linefeed sequence (CRLF).

The **GetFileType** method can be used to determine the current file type.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[GetFileType](#), [OpenFile](#), [RegisterFileType](#), [SetFileMode](#), [SetFileStructure](#), [SetPassiveMode](#)

CFtpClient::SetLastError Method

```
VOID SetLastError(  
    DWORD dwErrorCode  
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or FTP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

See Also

[GetErrorString](#), [GetLastError](#), [ShowError](#)

CFtpClient::SetPassiveMode Method

```
INT SetPassiveMode(  
    BOOL bPassiveMode  
);
```

The **SetPassiveMode** method enables or disables passive mode file transfers for the specified client session.

Parameters

bPassiveMode

A boolean flag which specifies that the client should enter passive mode and establish all connections with the server to transfer data.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

By default, the File Transfer Protocol uses active mode transfers, whereby the data connection is established from the server back to the local client. However, this can introduce problems for a client application that is behind a proxy server, firewall or a router which uses Network Address Translation (NAT). Enabling passive mode transfers instructs the client to create an outbound connection from the local system to the server for the data connection, similarly to how the control connection is established.

Not all servers may support passive mode, in which case an error will be returned to the client when this method is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [GetData](#), [GetFile](#), [ProxyConnect](#), [PutData](#), [PutFile](#)

CFtpClient::SetPriority Method

```
INT SetPriority(  
    INT nPriority  
);
```

The **SetPriority** method specifies the priority for file transfers.

Parameters

nPriority

An integer value which specifies the new priority for file transfers. It may be one of the following values:

Constant	Description
FTP_PRIORITY_NORMAL	The default priority which balances resource utilization and transfer speed. It is recommended that most applications use this priority.
FTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the transfer. It is typically used with worker threads running in the background when the amount of time required to perform the transfer is not critical.
FTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the transfer and meters the bandwidth allocated for the transfer. This priority will increase the average amount of time required to complete a file transfer.
FTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the transfer, allocating more memory for internal buffering. It can be used when it is important to transfer the file quickly, and there are no other threads currently performing file transfers at the time.
FTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization while attempting to transfer the file as quickly as possible. If the file transfer is being performed in the main UI thread, this priority can cause the application to appear to become non-responsive. No events will be generated during the transfer.

Return Value

If the method succeeds, the return value is the previous file transfer priority. If the method fails, the return value is `FTP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **SetPriority** method can be used to control the processor usage, memory and network bandwidth allocated for file transfers. The default priority balances resource utilization and transfer speed while ensuring that a single-threaded application remains responsive to the user. Lower priorities reduce the overall resource utilization at the expense of transfer speed. For example, if

you create a worker thread to download a file in the background and want to ensure that it has a minimal impact on the process, the `FTP_PRIORITY_BACKGROUND` value can be used.

Higher priority values increase the memory allocated for the transfers and increases processor utilization for the transfer. The `FTP_PRIORITY_CRITICAL` priority maximizes transfer speed at the expense of system resources. It is not recommended that you increase the file transfer priority unless you understand the implications of doing so and have thoroughly tested your application. If the file transfer is being performed in the main UI thread, increasing the priority may interfere with the normal processing of Windows messages and cause the application to appear to become non-responsive. It is also important to note that when the priority is set to `FTP_PRIORITY_CRITICAL`, normal progress events will not be generated during the transfer.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetPriority](#)

CFtpClient::SetTimeout Method

```
INT SetTimeout(  
    UINT nTimeout  
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[Connect](#), [GetTimeout](#), [IsReadable](#), [IsWritable](#), [Read](#), [Write](#)

CFtpClient::ShowError Method

```
INT ShowError(  
    LPCTSTR lpszAppTitle,  
    UINT uType,  
    DWORD dwErrorCode  
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A pointer to a string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** function in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that function for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetErrorString](#), [GetLastError](#), [SetLastError](#)

CFtpClient::UploadFile Method

```
BOOL UploadFile(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszFileURL,  
    UINT nTimeout  
    DWORD dwOptions  
    LPFTPTRANSFERSTATUS lpStatus  
    FTPEVENTPROC lpEventProc  
    DWORD_PTR dwParam  
);
```

The **UploadFile** method uploads the specified file from the local system to the server.

Parameters

lpszLocalFile

A pointer to a string that specifies the file on the local system that will be uploaded to the server. The file pathing and name conventions must be that of the local host.

lpszFileURL

A pointer to a string that specifies the complete URL of the file that will be created or overwritten on the server. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
FTP_OPTION_PASSIVE	This option specifies the client should attempt to establish the data connection with the server. When the client uploads or downloads a file, normally the server establishes a second connection back to the client which is used to transfer the file data. However, if the local system is behind a firewall or a NAT router, the server may not be able to create the data connection and the transfer will fail. By specifying this option, it forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it

	returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should use the AUTH command to negotiate an explicit secure connection. Some servers may only require this when connecting to the server on ports other than 990.

lpStatus

A pointer to an FTPTRANSFERSTATUS structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **UploadFile** method provides a convenient way for an application to upload a file in a single function call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session, change the current working directory if necessary and then upload the file to the server. The URL must be complete, and specify either a standard or secure FTP scheme:

```
[ftp|ftps|sftp]://[username : password] @[remotehost] [:remoteport] /
[path / ...] [filename] [;type=a|i]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. If a path is specified as part of the URL, then function will attempt to change the current working directory. Note that the path in an FTP URL is relative to the home directory of the user account and is not an absolute path starting at the root directory on the server. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "ftp" scheme is used and the FTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, either the "ftps" or "sftp" scheme must be specified.

The optional "type" value at the end of the file name determines if the file should be uploaded as a text or binary file. A value of "a" specifies that the file should be uploaded as a text file. A value of "i" specifies that the file should be uploaded as a binary file. If the type is not explicitly specified, the file will be uploaded as a binary file.

The *lpStatus* parameter can be used by the application to determine the final status of the

transfer, including the total number of bytes copied, the amount of time elapsed and other information related to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The *lpEventProc* parameter specifies a pointer to a function which will be periodically called during the file transfer process. This can be used to check the status of the transfer by calling **GetTransferStatus** and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The *dwParam* parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the *this* pointer as the value, and then cast it back to an object pointer inside the callback function. If no event handler is required, then a NULL pointer can be specified as the value for *lpEventProc* and the *dwParam* parameter will be ignored.

The **UploadFile** method is designed to provide a simpler interface for uploading a file. However, complex connections such as those using a proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **Connect** and then use **PutFile** to upload the file.

Example

```
CFtpClient ftpClient;
CString strLocalFile = _T("c:\\temp\\database.mdb");
CString strFileURL =
_T("ftp://update:secret@ftp.example.com/updates/database.mdb");

if (!ftpClient.UploadFile(strLocalFile, strFileURL))
{
    ftpClient.ShowError();
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpEventProc](#), [DownloadFile](#), [GetTransferStatus](#), [PutFile](#), [FTPTRANSFERSTATUS](#)

CFtpClient::ValidateUrl Method

```
BOOL ValidateUrl(  
    LPCTSTR lpszURL  
);
```

The **ValidateUrl** method determines if a string represents a valid FTP URL.

Parameters

lpszURL

A pointer to a string that specifies the URL to validate.

Return Value

If the specified URL is valid and the host name can be resolved to an IP address, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **ValidateUrl** method will check the value of a string to ensure that it represents a complete, valid URL using either a standard or secure FTP scheme. This method will not establish a connection with the server to verify that it exists, it will only attempt to resolve the host name to an IP address. If the remote host is specified as an IP address, this method will check to make sure that the address is formatted correctly. Note that if you wish to specify an IPv6 address, you must enclose the address in brackets.

To establish a connection with a server using a URL, use the **ConnectUrl** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ConnectUrl](#), [DownloadFile](#), [UploadFile](#)

CFtpClient::VerifyFile Method

```
BOOL VerifyFile(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszRemoteFile,  
    DWORD dwOptions  
);
```

The **VerifyFile** method attempts to verify that the contents of a file on the local system are the same as the specified file on the server.

Parameters

lpszLocalFile

A pointer to a string that specifies the name of file on the local system.

lpszRemoteFile

A pointer to a string that specifies the name of the file on the server.

dwOptions

Specifies the options that may be used when comparing the files. This parameter may be one or more of the following values:

Value	Description
FTP_VERIFY_DEFAULT	File verification should use the best option available based on the available server features. If the server supports the XMD5 command, the class will calculate an MD5 hash of the local file contents and compare the value with the file on the server. If the server does not support the XMD5 command, but it does support the XCRC command, the class will calculate a CRC32 checksum of the local file contents and compare the value with the file on the server. If the server does not support either the XMD5 or XCRC commands, the class will compare the size of the local and remote files.
FTP_VERIFY_SIZE	Files are verified by comparing the number of bytes of data in the local and remote files. This is the least reliable method, and should only be used if the server does not support either the XMD5 or XCRC commands.
FTP_VERIFY_CRC32	Files are verified by calculating a CRC32 checksum of the local file contents and comparing it with the value returned by the server in response to the XCRC command. This method should only be used if the server does not support the XMD5 command.
FTP_VERIFY_MD5	Files are verified by calculating an MD5 hash of the local file contents and comparing it with the value returned by the server in response to the XMD5 command. This is the preferred method for performing file verification.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **VerifyFile** method will attempt to verify that the contents of the local and remote files are identical using one of several methods, based on the features that the server supports. Preference will be given to the most reliable method available, using either an MD5 hash, a CRC32 checksum or comparing the size of the file, in that order.

It is not recommended that you use this method with text files because of the different end-of-line conventions used by different operating systems. For example, a text file on a Windows system uses a carriage-return and linefeed pair to indicate the end of a line of text. However, on a UNIX system, a single linefeed is used to indicate the end of a line. This can cause the **VerifyFile** method to indicate the files are not identical, even though the only difference is in the end-of-line characters that are used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DeleteFile](#), [GetFile](#), [PutFile](#)

CFtpClient::Write Method

```
INT Write(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Write(  
    LPCTSTR lpszBuffer,  
    INT cbBuffer  
);
```

The **Write** method sends the specified number of bytes to the server.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server. In an alternate form of the method, the pointer is to a string.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero, unless a pointer to a string is passed as the buffer argument. In that case, if the value is -1, all of the characters in the string, up to but not including the terminating null character, will be sent to the server.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the next asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableEvents](#), [IsBlocking](#), [IsReadable](#), [IsWritable](#), [Read](#), [RegisterEvent](#)

File Transfer Protocol Data Structures

- FTPCLIENTQUOTA
- FTPFILESTATUS
- FTPFILESTATUSEX
- FTPTRANSFERSTATUS
- FTPTRANSFERSTATUSEX
- SECURITYCREDENTIALS
- SECURITYINFO
- SYSTEMTIME

FTPCLIENTQUOTA Structure

This structure is used by the [GetClientQuota](#) method to return information about the file quota for the current client session.

```
typedef struct _FTPCLIENTQUOTA
{
    DWORD    dwFileCount;
    DWORD    dwFileLimit;
    DWORD    dwDiskUsage;
    DWORD    dwDiskLimit;
} FTPCLIENTQUOTA, *LPFTPCLIENTQUOTA;
```

Members

dwFileCount

An unsigned integer value which specifies the number of files that the user has created. If file quotas have not been enabled for the current user, this value will be zero.

dwFileLimit

An unsigned integer value which specifies the maximum number of files that may be created by the user. If file quotas have not been enabled for the current user, this value will be zero.

dwDiskUsage

An unsigned integer value which specifies the number of bytes of disk storage that has been allocated by the current user. If file quotas have not been enabled for the current user, this value will be zero.

dwDiskLimit

An unsigned integer value which specifies the maximum number of bytes of disk storage that may be allocated by the current user. If file quotas have not been enabled for the current user, this value will be zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

FTPFILESTATUS Structure

This structure is used by the **EnumFiles**, **GetFirstFile** and **GetNextFile** methods to return information about files on the server.

```
typedef struct _FTPFILESTATUS
{
    TCHAR          szFileName[FTP_MAXFILENAMELEN];
    TCHAR          szFileOwner[FTP_MAXOWNERNAMELEN];
    TCHAR          szFileGroup[FTP_MAXGROUPNAMELEN];
    BOOL           bIsDirectory;
    DWORD          dwFileSize;
    DWORD          dwFileLinks;
    DWORD          dwFileVersion;
    DWORD          dwFilePerms;
    SYSTEMTIME     stFileDate;
} FTPFILESTATUS, *LPFTPFILESTATUS;
```

Members

szFileName

A string buffer which contains the name of the file on the server.

szFileOwner

A string buffer which contains the name of the user that owns the file on the server. Note that not all server types support the concept of file ownership by a user. Some UNIX systems will not provide this information if an anonymous login was used. For the proprietary Sterling directory formats, the "mailbox" is stored in this member.

szFileGroup

A string buffer which contains the name of the group that owns the file on the server. Note that not all server types support the concept of file ownership by a group. For the proprietary Sterling directory formats, the "batch number" is stored in this member, with the character # prepended for the format FTP_DIRECTORY_STERLING_2.

bIsDirectory

A boolean flag which specifies if the file is actually a subdirectory.

dwFileSize

The size of the file in bytes on the server. Servers that return file information in an MS-DOS format will always set this value to zero if the file refers to a subdirectory. If the file is a text file, the file size on the server may be different than the size on the local host if different end-of-line character conventions are used. It should be noted that under VMS, the file size is reported in 512 byte blocks, so the size should be considered approximate on that platform.

dwFileLinks

The number of links to the file. Note that not all server types support the concept of file links, in which case this value will be zero.

dwFileVersion

The number of revisions made to the file. Note that not all server types support the concept of file versioning, in which case this value will be zero. Currently this value will only be non-zero for VMS platforms.

dwFilePerms

The permissions associated with the file. This value is actually a combination of bits that specify

the individual permissions for the file owner, group and world (all other users). For those familiar with UNIX, the file permissions are the same as those used by the `chmod` command. For the proprietary Sterling directory formats, a bit map representing the status codes and transfer protocol of the file is stored in this member.

stFileDate

A `SYSTEMTIME` structure which specifies the date that the file was created or last modified.

File Permissions

Constant	Description
<code>FILE_OWNER_READ</code>	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.
<code>FILE_OWNER_WRITE</code>	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
<code>FILE_OWNER_EXECUTE</code>	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
<code>FILE_GROUP_READ</code>	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
<code>FILE_GROUP_WRITE</code>	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
<code>FILE_GROUP_EXECUTE</code>	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
<code>FILE_WORLD_READ</code>	All users have permission to open the file for reading. This permission grants any user the right to download the file to the local system.
<code>FILE_WORLD_WRITE</code>	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.
<code>FILE_WORLD_EXECUTE</code>	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.

Sterling Status Codes

Bits 0-25 correspond to letters of the alphabet, most of which have distinct meanings in the Sterling formats.

Letter code	Bit position	Hexadecimal value
-------------	--------------	-------------------

A	0	1h
B	1	2h
C	2	4h
<i>n-th letter of alphabet</i>	n-1	2 to the (n-1) power
Z	25	2000000h

For the proprietary Sterling directory formats, bits 26-31 represent the transfer protocol associated with the file:

Protocol	Bit position	Hexadecimal value	Constant
TCP	26	4000000h	FTP_STERLING_STATUS_TCP
FTP	27	8000000h	FTP_STERLING_STATUS_FTP
BSC	28	10000000h	FTP_STERLING_STATUS_BSC
ASC	29	20000000h	FTP_STERLING_STATUS_ASC
FTS	30	40000000h	FTP_STERLING_STATUS_FTS
other	31	80000000h	FTP_STERLING_STATUS_OTHER

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

FTPFILESTATUSEX Structure

This structure is used by the **EnumFiles**, **GetFirstFile** and **GetNextFile** methods to return information about files on the server. This structure is designed for use with extended functions that support files larger than 4GB.

```
typedef struct _FTPFILESTATUSEX
{
    TCHAR          szFileName[FTP_MAXFILENAMELEN];
    TCHAR          szFileOwner[FTP_MAXOWNERNAMELEN];
    TCHAR          szFileGroup[FTP_MAXGROUPNAMELEN];
    BOOL           bIsDirectory;
    ULARGE_INTEGER uiFileSize;
    DWORD          dwFileLinks;
    DWORD          dwFileVersion;
    DWORD          dwFilePerms;
    DWORD          dwFileFlags;
    SYSTEMTIME     stFileDate;
} FTPFILESTATUSEX, *LPFTPFILESTATUSEX;
```

Members

szFileName

A string buffer which contains the name of the file on the server.

szFileOwner

A string buffer which contains the name of the user that owns the file on the server. Note that not all server types support the concept of file ownership by a user. Some UNIX systems will not provide this information if an anonymous login was used. For the proprietary Sterling directory formats, the "mailbox" is stored in this member.

szFileGroup

A string buffer which contains the name of the group that owns the file on the server. Note that not all server types support the concept of file ownership by a group. For the proprietary Sterling directory formats, the "batch number" is stored in this member, with the character # prepended for the format FTP_DIRECTORY_STERLING_2.

bIsDirectory

A boolean flag which specifies if the file is actually a subdirectory.

uiFileSize

The size of the file in bytes on the server. Servers that return file information in an MS-DOS format will always set this value to zero if the file refers to a subdirectory. If the file is a text file, the file size on the server may be different than the size on the local host if different end-of-line character conventions are used. It should be noted that under VMS, the file size is reported in 512 byte blocks, so the size should be considered approximate on that platform.

dwFileLinks

The number of links to the file. Note that not all server types support the concept of file links, in which case this value will be zero.

dwFileVersion

The number of revisions made to the file. Note that not all server types support the concept of file versioning, in which case this value will be zero. Currently this value will only be non-zero for VMS platforms.

dwFilePerms

The permissions associated with the file. This value is actually a combination of bits that specify the individual permissions for the file owner, group and world (all other users). For those familiar with UNIX, the file permissions are the same as those used by the `chmod` command. For the proprietary Sterling directory formats, a bit map representing the status codes and transfer protocol of the file is stored in this member.

dwFileFlags

This structure member is reserved for future use.

stFileDate

A [SYSTEMTIME](#) structure which specifies the date that the file was created or last modified.

File Permissions

Constant	Description
FILE_OWNER_READ	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.
FILE_OWNER_WRITE	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_OWNER_EXECUTE	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_GROUP_READ	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
FILE_GROUP_WRITE	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_GROUP_EXECUTE	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_WORLD_READ	All users have permission to open the file for reading. This permission grants any user the right to download the file to the local system.
FILE_WORLD_WRITE	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.
FILE_WORLD_EXECUTE	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.

Sterling Status Codes

Bits 0-25 correspond to letters of the alphabet, most of which have distinct meanings in the Sterling formats.

Letter code	Bit position	Hexadecimal value
A	0	1h
B	1	2h
C	2	4h
<i>n-th letter of alphabet</i>	n-1	2 to the (n-1) power
Z	25	2000000h

For the proprietary Sterling directory formats, bits 26-31 represent the transfer protocol associated with the file:

Protocol	Bit position	Hexadecimal value	Constant
TCP	26	4000000h	FTP_STERLING_STATUS_TCP
FTP	27	8000000h	FTP_STERLING_STATUS_FTP
BSC	28	10000000h	FTP_STERLING_STATUS_BSC
ASC	29	20000000h	FTP_STERLING_STATUS_ASC
FTS	30	40000000h	FTP_STERLING_STATUS_FTS
other	31	80000000h	FTP_STERLING_STATUS_OTHER

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

FTPTRANSFERSTATUS Structure

This structure is used by the **GetTransferStatus** method to return information about a file transfer in progress.

```
typedef struct _FTPTRANSFERSTATUS
{
    DWORD    dwBytesTotal;
    DWORD    dwBytesCopied;
    DWORD    dwBytesPerSecond;
    DWORD    dwTimeElapsed;
    DWORD    dwTimeEstimated;
    TCHAR    szLocalFile[FTP_MAXFILENAMELEN];
    TCHAR    szRemoteFile[FTP_MAXFILENAMELEN];
} FTPTRANSFERSTATUS, *LPFTPTRANSFERSTATUS;
```

Members

dwBytesTotal

The total number of bytes that will be transferred. If the file is being copied from the server to the local host, this is the size of the remote file. If the file is being copied from the local host to the server, it is the size of the local file. If the file size cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the file transfer is completed. This is based on the average number of bytes transferred per second.

szLocalFile

A pointer to a string which specifies the local file that is being copied to or from the server.

szRemoteFile

A pointer to a string which specifies the remote file that is being copied to or from the local system.

Remarks

If the option `FTP_OPTION_HIRES_TIMER` has been specified when connecting to the server, the values returned in the *dwTimeElapsed* and *dwTimeEstimated* members will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small files over a fast network connection.

If you are uploading or downloading large files which exceed 4GB, you should use the **FTPTRANSFERSTATUSEX** structure which uses 64-bit integers for the file size.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

See Also

[GetTransferStatus](#), [FTPTRANSFERSTATUSEX](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

FTPTRANSFERSTATUSEX Structure

This structure is used by the **GetTransferStatus** method to return information about a file transfer in progress. This structure is designed for use with extended functions that support files larger than 4GB.

```
typedef struct _FTPTRANSFERSTATUSEX
{
    ULARGE_INTEGER dwBytesTotal;
    ULARGE_INTEGER dwBytesCopied;
    DWORD          dwBytesPerSecond;
    DWORD          dwTimeElapsed;
    DWORD          dwTimeEstimated;
    DWORD          dwReserved;
    TCHAR          szLocalFile[FTP_MAXFILENAMELEN];
    TCHAR          szRemoteFile[FTP_MAXFILENAMELEN];
} FTPTRANSFERSTATUSEX, *LPFTPTRANSFERSTATUSEX;
```

Members

uiBytesTotal

The total number of bytes that will be transferred. If the file is being copied from the server to the local host, this is the size of the remote file. If the file is being copied from the local host to the server, it is the size of the local file. If the file size cannot be determined, this value will be zero.

uiBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the file transfer is completed. This is based on the average number of bytes transferred per second.

dwReserved

This structure member is reserved for future use.

szLocalFile

A pointer to a string which specifies the local file that is being copied to or from the server.

szRemoteFile

A pointer to a string which specifies the remote file that is being copied to or from the local system.

Remarks

If the option `FTP_OPTION_HIRES_TIMER` has been specified when connecting to the server, the values returned in the *dwTimeElapsed* and *dwTimeEstimated* members will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small files over a fast network connection.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

See Also

[GetTransferStatus](#), [FTPTRANSFERSTATUS](#)

SECURITYCREDENTIALS Structure

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on

	<p>what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.</p>
SECURITY_PROTOCOL_TLS10	<p>The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS11	<p>The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS12	<p>The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.</p>
SECURITY_PROTOCOL_SSH	<p>Either version 1.0 or 2.0 of the Secure Shell protocol should be used when establishing the connection. The correct protocol is automatically selected based on the version of the protocol that is supported by the server.</p>
SECURITY_PROTOCOL_SSH1	<p>The Secure Shell 1.0 protocol should be used when establishing the connection. This is an older version of the protocol which should not be used unless explicitly required by the server. Most modern SSH server support version 2.0 of the protocol.</p>
SECURITY_PROTOCOL_SSH2	<p>The Secure Shell 2.0 protocol should be used when establishing the connection. This is the default version of the protocol that is supported by most SSH servers.</p>

dwOptions

A value which specifies one or options. This value should always be zero for connections using SSH. This member is constructed by using a bitwise operator with any of the following values:



Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A pointer to a null terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A pointer to a null terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A pointer to a null terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A pointer to a null terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
------------	-------------

CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
----	---

MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
----	--

ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are
------	---

installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

A pointer to a null terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used for SSH connections and should always be NULL when establishing a secure connection using SSL or TLS.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* member should be set to CREDENTIAL_STORE_FILENAME and the *lpszCertStore* member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS#12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the *lpszPassword* member or the library will be unable to access the certificate.

Note that the *lpszUserName* and *lpszPassword* members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

The TLS 1.1 and TLS 1.2 protocols are only supported on Windows 7, Windows Server 2008 R2 and later versions of the platform. If these options are specified and the application is running on Windows XP or Windows Vista, the protocol version will be downgraded to TLS 1.0 for backwards compatibility with those platforms.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

SECURITYINFO Structure

This structure contains information about a secure connection that has been established with a server.

```
typedef struct _SECURITYINFO
{
    DWORD        dwSize;
    DWORD        dwProtocol;
    DWORD        dwCipher;
    DWORD        dwCipherStrength;
    DWORD        dwHash;
    DWORD        dwHashStrength;
    DWORD        dwKeyExchange;
    DWORD        dwCertStatus;
    SYSTEMTIME   stCertIssued;
    SYSTEMTIME   stCertExpires;
    LPCTSTR      lpszCertIssuer;
    LPCTSTR      lpszCertSubject;
    LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to **sizeof(SECURITYINFO)** prior to passing the address of this structure to the function. Note that if this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the protocol that was selected to establish the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be

	<p>used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.</p>
SECURITY_PROTOCOL_TLS10	<p>The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS11	<p>The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS12	<p>The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.</p>
SECURITY_PROTOCOL_SSH1	<p>The Secure Shell 1.0 protocol has been selected. This protocol has been deprecated and is no longer widely used. It is not recommended that this protocol be used when establishing secure connections. This protocol can only be specified when connecting to an SSH server and is not supported with any other application protocol.</p>
SECURITY_PROTOCOL_SSH2	<p>The Secure Shell 2.0 protocol has been selected. This is the most commonly used version of the protocol. It is recommended that this version of the protocol be used unless the server explicitly requires the client to use an earlier version. This protocol can only be specified when connecting to an SSH server and is not supported with any other application protocol.</p>

A numeric value which specifies the cipher that was selected when establishing the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_CIPHER_RC2	The RC2 block cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
SECURITY_CIPHER_DES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher (also known as the Rijndael cipher) is a fixed block size cipher which use a key size of 128, 192 or 256 bits. This cipher is supported on Windows XP SP3 and later versions of the operating system.
SECURITY_CIPHER_SKIPJACK	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
SECURITY_CIPHER_BLOWFISH	The Blowfish block cipher was selected. This is a variable key length cipher up to 448 bits, using a 64-bit block size.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected. One of the following values will be returned:

Constant	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.

SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected. One of the following values will be returned:

Constant	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography. This key exchange algorithm is only supported on Windows XP SP3 and later versions of the operating system.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server. This member only has meaning for connections using the SSL or TLS protocols. One of the following values will be returned:

Constant	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertIssuer

A pointer to a string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertSubject

A pointer to a string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszFingerprint

A pointer to a string which contains a sequence of hexadecimal values that uniquely identify the server. This member is only used when a connection has been established using the Secure Shell (SSH) protocol.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

SYSTEMTIME Structure

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include windows.h.

File Transfer Protocol Server Class

Implements a server that enables the application to send and receive files using the File Transfer Protocol.

Reference

- [Data Members](#)
- [Class Methods](#)
- [Event Handlers](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

File Name	CSFTSV10.DLL
Version	10.0.1468.2518
LibID	5CED7337-69F7-4662-B173-42FA4EEB64E3
Import Library	CSFTSV10.LIB
Dependencies	None
Standards	RFC 959, RFC 1579, RFC 2228

Overview

This library provides an interface for implementing an embedded, lightweight server that can be used to exchange files with a client using the standard File Transfer Protocol. The server can accept connections from any third-party application or a program developed using the SocketTools FTP client API.

The application specifies an initial server configuration and then responds to events that are raised by the API when the client sends a request to the server. An application may implement only minimal handlers for most events, in which case the default actions are performed for most standard FTP commands. However, an application may also use the event mechanism to filter specific commands or to extend the protocol by providing custom implementations of existing commands or add entirely new commands.

The server supports active and passive mode file transfers, has compatibility options for NAT router and firewall support, and provides support for secure file transfers using explicit TLS. Secure connections require a valid security certificate to be installed on the system.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical

updates available for your version of the operating system.

This class provides an implementation of a multithreaded server which should only be used with languages that support the creation of multithreaded applications. It is important that you do not attempt to link against static libraries which were not built with support for threading.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

CFtpServer Public Data Members

Member Variables	Description
m_dwOptions	Options specified when creating an instance of the server
m_nAuthFail	The maximum number of user authentication failures permitted per client session
m_nAuthTime	The maximum number of seconds a client has to successfully authenticate the session
m_nExecTime	The maximum number of seconds that the server will permit an external command to execute
m_nIdleTime	The maximum number of seconds a client can be idle before the server terminates the session
m_nLogFormat	The format used by the server to log client activity
m_nLogLevel	The level of detail included in the server log file
m_nMaxClients	The maximum number of active client sessions accepted by the server
m_nMaxClientsPerAddress	The maximum number of clients per IP address accepted by the server
m_nMaxGuests	The maximum number of anonymous client sessions accepted by the server
m_nMaxPort	The maximum port number used by the server for passive data connections
m_nMinPort	The minimum port number used by the server for passive data connections

CFtpServer::m_dwOptions

DWORD m_dwOptions;

The default options used when starting an instance of the server.

Remarks

The **m_dwOptions** data member is a public variable that specifies the default options that should be used when starting an instance of the server. This variable can be modified directly or by calling the **SetOptions** method. For a list of available server options, see [Server Option Constants](#). Changing the value of this data member does not have an effect on an active instance of the server.

See Also

[CFtpServer](#), [GetOptions](#), [SetOptions](#)

CFtpServer::m_nAuthFail

`UINT m_nAuthFail;`

The maximum number of user authentication failures permitted per client session.

Remarks

The **m_nAuthFail** data member is a public variable that specifies the maximum number of user authentication attempts that are permitted until the server terminates the client connection. A value of zero specifies that the default configuration limit of 3 authentication attempts per login should be allowed. The maximum number of authentication attempts is 10. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

[CFtpServer](#)

CFtpServer::m_nAuthTime

```
UINT m_nAuthTime;
```

The maximum number of seconds a client has to successfully authenticate the session.

Remarks

The **m_nAuthTime** data member is a public variable that specifies the maximum number of seconds that a client session must authenticate itself. A value of zero specifies the default value of 60 seconds. If the value is non-zero, the minimum value is 20 seconds and the maximum value is 300 seconds (5 minutes). This value is used to ensure that a client has successfully authenticated itself within a limited period of time.

This time limit prevents a potential denial-of-service attack against the server where clients establish connections and hold them open without authentication. In conjunction with the **m_nAuthFail** data member, this also limits the ability of a client to attempt to probe the server for valid username and password combinations. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

[CFtpServer](#)

CFTPServer::m_nExecTime

`UINT m_nExecTime;`

The maximum number of seconds that the server will permit an external command to execute.

Remarks

The **m_nExecTime** data member is a public variable that specifies the maximum number of seconds that an external program is permitted to run on the server. External programs are registered using the **RegisterProgram** method, and are executed by the client sending the SITE EXEC command to the server. If this value is zero, the default timeout period of 5 seconds will be used. The minimum execution time is 1 second and the maximum time limit is 30 seconds. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

[CFTPServer](#), [RegisterProgram](#)

CFtpServer::m_nIdleTime

`UINT m_nIdleTime;`

The maximum number of seconds a client can be idle before the server terminates the session.

Remarks

The **m_nIdleTime** data member is a public variable that specifies the maximum number of seconds that a client session may be idle before the server closes the control connection to the client. A value of zero specifies the default value of 900 seconds (15 minutes). If the value is non-zero, the minimum value is 60 seconds and the maximum value is 7200 seconds (2 hours). This value is used to initialize the default idle timeout period for each client session. A client may request that the server change the idle timeout period for its session by sending the SITE IDLE command. The server determines if a client is idle based on the time the last command was issued and whether or not a file transfer is in progress. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

[CFtpServer](#), [RegisterProgram](#)

CFtpServer::m_nLogFormat

`UINT m_nLogFormat;`

The format used when updating the server log file.

Remarks

The **m_nLogFormat** data member is a public variable that specifies the format of the log file that is created or updated by the server. It may be one of the following values:

Constant	Description
FTP_LOGFILE_NONE (0)	This value specifies that the server should not create or update a log file.
FTP_LOGFILE_COMMON (1)	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about file transfers.
FTP_LOGFILE_EXTENDED (2)	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session.

By default, logging is not enabled for the server. Changing the value of this data member does not have an effect on an active instance of the server. To change the format, level of detail or default log file name, use the **SetLogFile** method.

See Also

[CFtpServer](#), [GetLogFile](#), [SetLogFile](#)

CFtpServer::m_nLogLevel

UINT m_nLogLevel;

The level of detail included in the server log file.

Remarks

The **m_nLogLevel** data member is a public variable that specifies the level of detail that should be generated in the log file. The minimum value is 1 and the maximum value is 10. If the **m_nLogFormat** data member specifies a valid log file format and this value is zero, a default level of detail will be selected based on the format.

The common log file format generally contains less information by default, only logging the data transfers between the client and server. The W3C extended log file format defaults to a higher level of detail that includes additional information about the client session. The higher the level of detail, the larger the log file will be.

By default, logging is not enabled for the server. Changing the value of this data member does not have an effect on an active instance of the server. To change the format, level of detail or default log file name, use the **SetLogFile** method.

See Also

[CFtpServer](#), [GetLogFile](#), [SetLogFile](#)

CFTPServer::m_nMaxClients

`UINT m_nMaxClients;`

The maximum number of clients that are permitted to connect to the server.

Remarks

The **m_nMaxClients** data member is a public variable that specifies the maximum number of clients that are permitted to establish a connection with the server. After this limit is reached, the server will reject additional connections until the number of active clients drops below this threshold. A value of zero specifies that there is no fixed limit on the active number of client connections. Changing the value of this data member does not have an effect on an active instance of the server. To change the maximum number of clients on an active server, use the **Throttle** method.

The actual number of client connections that can be accepted depends on the amount of memory available to the server process. Sockets are allocated from the non-paged memory pool, so the actual number of sockets that can be created system-wide depends on the amount of physical memory that is installed. If the server will be accessible over the Internet, it is recommended that you limit the maximum number of client connections to a reasonable value.

See Also

[CFTPServer](#), [Throttle](#)

CFtpServer::m_nMaxClientsPerAddress

`UINT m_nMaxClientsPerAddress;`

The maximum number of clients that are permitted to connect to the server from a single IP address.

Remarks

The **m_nMaxClientsPerAddress** data member is a public variable that specifies the maximum number of clients that are permitted to establish a connection with the server from a single IP address. After this limit is reached, the server will reject additional connections until the number of active clients drops below this threshold. A value of zero specifies that there is no limit on the active number of client connections per IP address. Changing the value of this data member does not have an effect on an active instance of the server. To change the maximum number of clients on an active server, use the **Throttle** method.

See Also

[CFtpServer](#), [Throttle](#)

CFTPServer::m_nMaxGuests

`UINT m_nMaxGuests;`

The maximum number of anonymous users that are permitted to connect to the server.

Remarks

The **m_nMaxGuests** data member is a public variable that specifies the maximum number of anonymous users that are permitted to establish a connection with the server. After this limit is reached, the server will send an error response indicating that the server is unavailable, and then immediately terminate the session. Unlike the limit on the total number of client connections, this limit is only checked after the client has requested authentication and has logged in as an anonymous user. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

[CFTPServer](#)

CFtpServer::m_nMaxPort

UINT m_nMaxPort;

The maximum port number used by the server for passive data connections.

Remarks

The **m_nMaxPort** data member is a public variable that specifies the maximum range of port numbers that will be used with passive data connections. A value of zero specifies the default value of 65535 should be used. The minimum value of this member is 5000 and the maximum value is 65535. If the value is non-zero, it must be greater than the value of the **m_nMinPort** data member. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

[CFtpServer](#)

CFTPServer::m_nMinPort

UINT m_nMinPort;

The minimum port number used by the server for passive data connections.

Remarks

The **m_nMinPort** data member is a public variable that specifies the minimum range of port numbers that will be used with passive data connections. A value of zero specifies that the default value of 30000 should be used. The minimum value of this member is 5000 and the maximum value is 65535. If the value is non-zero, it must be less than the value of the **m_nMaxPort** data member. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

[CFTPServer](#)

CFtpServer Methods

Class	Description
CFtpServer	Constructor which initializes the current instance of the class
~CFtpServer	Destructor which releases resources allocated by the class
Method	Description
AddVirtualUser	Add a new virtual user for the specified server
AsyncNotify	Enable or disable asynchronous notification of changes in server status
AttachHandle	Attach the specified server handle to this instance of the class
AuthenticateClient	Authenticate the client and assign access rights for the session
ChangeClientDirectory	Change the current working directory for the client session
DeleteVirtualUser	Delete a virtual user from the specified server
DetachHandle	Detach the server handle from the current instance of this class
DisableCommand	Disable a specific server command
DisableTrace	Disable the logging of network function calls
DisconnectClient	Disconnect the specific client session, closing the control channel and aborting any file transfer
EnableClientAccess	Enable or disable access rights for the specified client session
EnableCommand	Enable a specific server command
EnableTrace	Enable logging of network function calls to a file
EnumClients	Returns a list of active client connections established with the server
GetActiveClient	Return the client ID for the active client session associated with the current thread
GetAddress	Return the IP address for the server
GetClientAccess	Return the access rights that have been granted to the client session
GetClientAddress	Return the IP address of the specified client session
GetClientCredentials	Return the credentials for the specified client session
GetClientDirectory	Return the current working directory for a client session
GetClientFileType	Return the current file type used for transfers by the specified client
GetClientHomeDirectory	Return the home directory for an authenticated client session
GetClientIdentity	Return the identity of the specified client session
GetClientIdleTime	Return the idle timeout period for the specified client
GetClientLocalPath	Return the full local path for the specified virtual path
GetClientServer	Return the handle to the server that created the specified client session
GetClientThreadId	Returns the thread ID associated with the specified client session

GetClientUserName	Return the user name associated with the specified client session
GetClientVirtualPath	Return the virtual path for a local file on the server
GetCommandFile	Return the full path to the local file name or directory specified by the client
GetCommandLine	Return the complete command line issued by the client
GetCommandName	Return the name of the last command issued by the client
GetCommandParam	Return the value of the specified parameter for the command issued by the client
GetCommandParamCount	Return the number of parameters to the current command issued by the client
GetCommandResult	Return the result code and a description of the last command processed by the server
GetCommandUsage	Return the number of times a specific command has been issued by all clients
GetDirectory	Return the full path to the root directory assigned to the specified server
GetHandle	Return the server handle associated with the class instance
GetIdentity	Return the identity and version information for the specified server
GetLastError	Return information about the last server error that occurred
GetLogFile	Return the current log file format and full path for the file
GetMemoryUsage	Return the amount of memory allocated for the server and all client sessions
GetName	Return the host name assigned to the server or specified client session
GetOptions	Return the options specified for this instance of the server
GetPriority	Return the current priority assigned to the specified server
GetProgramExitCode	Return the exit code of the last program executed by the client
GetProgramName	Return the name of the last program executed by the client
GetProgramOutput	Return a copy of the standard output from the last program executed by the client
GetProgramText	Return a copy of the standard output from the last program in a string buffer
GetRenamedFile	Return the original name of a file being renamed by the client
GetStackSize	Return the initial size of the stack allocated for threads created by the server
GetTransferInfo	Return information about the current file transfer for the client session
GetUuid	Return the UUID assigned to the specified server
IsActive	Determine if the server has been started
IsClientAnonymous	Determine if the specified client has authenticated as an anonymous user
IsClientAuthenticated	Determine if the specified client session has been authenticated
IsCommandEnabled	Determine if the specified command is currently enabled or disabled
IsInitialized	Determine if the class has been successfully initialized

IsListening	Determine if the server is listening for client connections
PreProcessEvent	Filter server events before being processed by the default event handler
RegisterProgram	Register a program for use with the SITE EXEC command
RenameServerLogFile	Rename or delete the current log file being updated by the server
Restart	Restart the server, terminating all active client sessions
Resume	Resume accepting client connections on the specified server
SendResponse	Send a result code and optional message to the client in response to a command
SetAddress	Change the IP address that the server will use with passive data connections
SetCertificate	Set the name of the certificate to be used with secure connections.
SetClientAccess	Change the access rights associated with the specified client session
SetClientFileType	Change the current file type used for transfers by the specified client
SetClientIdentity	Change the identity string associated with the specified client session
SetClientIdleTime	Change the idle timeout period for the specified client session
SetCommandFile	Change the name of the local file or directory that is the target of the current command
SetDirectory	Specify the local directory that will be used as the server root directory
SetLastError	Set the last error code for the specified server session
SetLogFile	Change the current log format, level of detail and file name
SetIdentity	Change the identity and version information for the specified server
SetName	Change the hostname assigned to the server or specified client session
SetOptions	Change the options specified for this instance of the server
SetPriority	Change the priority assigned to the specified server
SetStackSize	Change the initial size of the stack allocated for threads created by the server
SetUuid	Assign a UUID to be associated with this instance of the server
Start	Start the server and begin accepting client connections
Stop	Stop the server and terminate all active client connections
Suspend	Suspend accepting client connections on the specified server
Throttle	Limit the number of active client connections, connections per address and connection rate

CFtpServer::~CFtpServer

`~CFtpServer();`

The **CFtpServer** destructor releases resources allocated by the current instance of the **CFtpServer** object. It also uninitialized the library if there are no other concurrent uses of the class.

Remarks

When a **CFtpServer** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CFtpServer](#)

CFtpServer::AddVirtualUser Method

```
BOOL AddVirtualUser(  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    DWORD dwUserAccess,  
    LPCTSTR lpszDirectory  
);
```

Add a new virtual user for the specified server.

Parameters

lpszUserName

A pointer to a string which specifies the user name. The maximum length of a username is 63 characters and it is recommended that names be limited to alphanumeric characters. Whitespace, control characters and certain symbols such as path delimiters and wildcard characters are not permitted. If an invalid character is included in the name, the method will fail with an error indicating the username is invalid. This parameter cannot be NULL and the name must be at least three characters in length. Usernames are not case sensitive.

lpszPassword

A pointer to a string which specifies the user password. The maximum length of a password is 63 characters and is limited to printable characters. Whitespace and control characters are not permitted. If an invalid character is included in the password, the method will fail with an error indicating the password is invalid. This parameter cannot be NULL and must be at least one character in length. Passwords are case sensitive.

dwUserAccess

An integer value which specifies the access clients will be given when authenticated as this user. For a list of user access permissions, see [User Access Constants](#).

lpszDirectory

A pointer to a string which specifies the directory that will be the virtual user's home directory. If the server was started in multi-user mode, this directory will be relative to the user directory created by the server, otherwise it will be relative to the server root directory. If the directory does not exist, it will be created the first time that the virtual user successfully logs in to the server. If this parameter is NULL or an empty string, a default home directory will be created for the virtual user.

Return Value

If the method succeeds, the return value is non-zero. If the username or password contain invalid characters, the method will return zero. If the method fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **AddVirtualUser** method adds a virtual user that is associated with the specified virtual host. When a client connects with the server and provides authentication credentials, the server will check if the username has been created using this method. If a match is found, the client access rights will be updated.

If you wish to modify the information for a user, it is not necessary to delete the username first. If this method is called with a username that already exists, that record is replaced with the values passed to this method. You cannot use this method to create a virtual user named "anonymous".

The virtual users created by this method exist only as long as the server is active. If you wish to maintain a persistent database of users and passwords, you are responsible for its implementation based on the requirements of your specific application. For example, a simple implementation would be to store the user information in a local XML or INI file and then read that configuration file after the server has started, calling this method for each user that is listed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DeleteVirtualUser](#)

CFtpServer::AsyncNotify Method

```
BOOL AsyncNotify(  
    HWND hWnd,  
    UINT uMsg  
);
```

Enable or disable asynchronous notification of changes in server status.

Parameters

hWnd

A handle to the window whose window procedure will receive the notification message.

uMsg

The user-defined message that will be sent to the notification window.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **AsyncNotify** method is used by an application to enable or disable asynchronous notifications. The message window is typically the main UI window and these notifications are used signal to the application that it should update the user interface. If the *hWnd* parameter is not NULL, it must specify a valid window handle and the user-defined message must have a value of **WM_USER** or higher. The application cannot specify a notification message that is reserved by the operating system. The pseudo-handle **HWND_BROADCAST** cannot be specified as the notification window. If the *hWnd* parameter is NULL, notifications for the specified server will be disabled.

When asynchronous notifications are enabled for a server, the server will post the user-defined message to the window whenever there is a change in status or after a client has connected or disconnected from the server. The *wParam* message parameter will contain the notification message and the *lParam* message parameter will contain the handle to the server or the client ID. The following notification messages are defined:

Constant	Description
FTP_NOTIFY_STARTUP	This notification is sent when the server has started and is preparing to accept client connections. This notification is only sent once, and only if asynchronous notifications are enabled immediately after the Start method is called. This message will not be sent once the server has begun accepting client connections or when notification messages are disabled and then subsequently re-enabled at a later time. The <i>lParam</i> message parameter will specify the handle to the server.
FTP_NOTIFY_LISTEN	This notification is sent when the server is listening for client connections. This notification message may be sent to the application multiple times over the lifetime of the server. If the server was suspended, this notification will be sent after the application calls the Resume method to

	resume accepting client connections. The <i>lParam</i> message parameter will specify the handle to the server.
FTP_NOTIFY_SUSPEND	This notification is sent when the server suspends accepting new connections because the application has called the Suspend method. This notification message may be sent to the application multiple times over the lifetime of the server. The <i>lParam</i> message parameter will specify the handle to the server.
FTP_NOTIFY_RESTART	This notification is sent when the server is restarted using the Restart method. Note that the server socket handle provided by the <i>lParam</i> message parameter will specify the new socket handle of the restarted server instance, not the original socket handle. The <i>lParam</i> message parameter will specify the handle to the server.
FTP_NOTIFY_CONNECT	This notification is sent when the server accepts a client connection and the thread that manages the client session has begun processing network events for that client. This message notification will not be sent if the client connection is rejected by the server. The <i>lParam</i> message parameter will specify the unique ID of the client that connected to the server.
FTP_NOTIFY_DISCONNECT	This notification is sent when the client disconnects from the server and the client socket has been closed. This notification message may not occur for each client session that is forced to terminate as the result of the server being stopped using the Stop method. The <i>lParam</i> message parameter will specify the unique ID of the client that disconnected from the server.
FTP_NOTIFY_SHUTDOWN	This notification is sent when the server thread is in the process of terminating. At the time the application processes this notification message, the server handle in <i>lParam</i> will reference the defunct server and cannot be used with other server methods. The <i>lParam</i> message parameter will specify the handle to the server.

If asynchronous notifications are enabled, you should never use those notifications as a replacement for an event handler. When an event occurs, the callback function that handles the event is invoked in the context of the thread that manages the client session. The application should exchange data with the client within that event handler and not in response to a notification message. These notification messages should only be used to update the application UI in response to changes in the status of the server.

The FTP_NOTIFY_CONNECT and FTP_NOTIFY_DISCONNECT notifications are different from the other server notifications because the *lParam* message parameter does not specify the server handle, but rather the unique client ID associated with the session that connected to or disconnected from the server. Use the **GetClientServer** method to obtain a handle to the server that created the client session. Note that at the time the application processes the FTP_NOTIFY_DISCONNECT notification message, the client session will have already terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cstools10.h.

Import Library: csftsv10.lib

See Also

[GetClientServer](#), [Start](#)

CFtpServer::AttachHandle Method

```
VOID AttachHandle(  
    HSERVER hServer  
);
```

The **AttachHandle** method attaches the specified server handle to the current instance of the class.

Parameters

hServer

The handle to the server that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a server handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that server.

If a server handle already has been created for the class, that handle will be released when the new handle is attached to the class object. This will cause the server to stop and all client sessions will be terminated immediately. If you want to prevent the previous server from being stopped, you must call the **DetachHandle** method prior to attaching a new handle to the class instance.

Note that the *hServer* parameter is presumed to be a valid server handle and no checks are performed to ensure that the handle references an active server. Specifying an invalid server handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[DetachHandle](#), [GetHandle](#)

CFtpServer::AuthenticateClient Method

```
BOOL AuthenticateClient(  
    UINT nClientId,  
    DWORD dwUserAccess,  
    BOOL bCreateHome,  
    LPCTSTR lpszDirectory  
);
```

Authenticate the client and assign access rights for the session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies one or more user access rights. For a list of user access rights that can be granted to the client, see [User Access Constants](#).

bCreateHome

An integer value that specifies if the server should create the home directory for the authenticated client if it does not already exist. If this value is non-zero, the home directory will be created. If value is zero, the home directory will not be created and if it does not exist, this function will fail.

lpszDirectory

A pointer to a string that specifies the home directory for the user. If an absolute path is specified, it will be relative to the server root directory. If a relative path is specified, it will be a subdirectory of the home directory for the server instance. If this parameter is NULL or an empty string, a home directory will be assigned based on the server home directory and the user name.

Return Value

If the the client session could be authenticated, the return value is non-zero. If the client ID does not specify a valid client session, or the client has already been authenticated, this method will return zero.

Remarks

The **AuthenticateClient** method is used to authenticate a specific client session, typically in response to an **OnAuthenticate** event that indicates a client has requested authentication. This method is also used internally to automatically grant the appropriate access rights to local user and anonymous client sessions.

It is recommended that most applications specify FTP_ACCESS_DEFAULT as the *dwUserAccess* value for a client session, since this allows the server automatically grant the appropriate access based on the server configuration options for normal and anonymous users. If the server is going to be publicly accessible or third-party FTP clients will be used to access the server, you should always grant the FTP_ACCESS_LIST permission to clients. Many client applications will not function correctly if they are unable to obtain a list of files in the user's home directory.

If FTP_ACCESS_RESTRICTED is specified and the server was started in multi-user mode, the client session will be effectively locked to its home directory and cannot navigate to the server root directory. By default, restricted client sessions are also limited to only downloading files and

requesting directory listings. If a client session is not restricted, the client can access files outside of its home directory. Regardless of this option, a client cannot access files outside of the server root directory.

If `FTP_ACCESS_RESTRICTED` or `FTP_ACCESS_ANONYMOUS` is specified, the client session will be authenticated in a restricted mode and the access rights for the session will persist until the client disconnects from the server. Unlike regular users, the access rights for a restricted client cannot be changed by the server at a later point. This restriction is designed to prevent the inadvertent granting of rights to an untrusted client that could compromise the security of the server.

If the *lpszDirectory* parameter is `NULL` or an empty string and the server has been started in multi-user mode, each user is assigned their own home directory based on their username. If the server has not been started in multi-user mode, then the default home directory will be the server root directory and is shared by all users. The **GetClientHomeDirectory** method will return the full path to the home directory for an authenticated client.

If the `FTP_ACCESS_EXECUTE` permission is granted to the client session, it can execute external programs using the `SITE EXEC` command. Because the program is executed in the context of the server process, it is recommended that you limit access to this functionality and ensure that the programs being executed do not introduce any security risks to the operating system. This permission is never granted by default, and the `SITE EXEC` command will return an error if the client session is anonymous, regardless of whether this permission is granted or not.

This method should only be used for custom authentication schemes and is not necessary if you have used the **AddVirtualUser** method to create virtual users.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AddVirtualUser](#), [ChangeClientDirectory](#), [GetClientCredentials](#), [GetClientDirectory](#), [OnAuthenticate](#)

CFtpServer::CFtpServer Method

`CFtpServer();`

The **CFtpServer** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **FtpServerInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** method because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[~CFtpServer](#), [IsInitialized](#)

CFtpServer::ChangeClientDirectory Method

```
BOOL ChangeClientDirectory(  
    UINT nClientId,  
    LPCTSTR lpszDirectory  
);
```

Change the current working directory for the specified client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszDirectory

A pointer to a string which specifies the new current working directory for the client session. If this parameter is NULL or an empty string, the current working directory will be changed to the client home directory. If this parameter is not NULL, it must specify a directory that exists and is accessible by the server process.

Return Value

If the current working directory was changed, the return value is non-zero. If the client ID does not specify a valid client session, or the directory is invalid, this method will return zero.

Remarks

The **ChangeClientDirectory** method will change the current working directory for the specified client session. This method is called internally when the client sends the CWD or CDUP commands, however it may be explicitly used by the application to change the client's working directory in response to a server event.

This method cannot be used to change the current working directory for a client to an arbitrary directory outside of the server root directory. If the *lpszDirectory* parameter specifies a relative path (i.e.: a path that does not begin with a drive letter or leading path delimiter) then the new working directory will be relative to the current working directory. If an absolute path is specified, the absolute path must include the complete path to either the server root directory or the user's home directory, based on the permissions granted to the client session. If a path outside of the server root directory is specified, this method will fail with an access denied error.

Use caution when calling this method to override the directory specified by the client when it sends the CWD or CDUP commands. If your application changes the current working directory to one not specified by the client, it may cause unpredictable behavior in the client application because the actual path of the current working directory will not match the directory that was requested.

If this method is used to change the current working directory in response to the CWD command, you should not call the **GetCommandParam** method and pass the command parameter as an argument to this method. You must use the **GetCommandFile** method to obtain the directory name provided by the client prior to calling this method.

The application should never call the **SetCurrentDirectory** function in the Windows API to change the current directory for the process to the working directory of a client session. Because the server is multithreaded and each client session is managed in its own thread, an application using this library should avoid using relative paths.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientDirectory](#), [GetClientHomeDirectory](#), [GetCommandFile](#)

CFtpServer::DeleteVirtualUser Method

```
BOOL DeleteVirtualUser(  
    UINT nHostId,  
    LPCTSTR LpszUserName  
);
```

Remove a virtual user from the specified host.

Parameters

nHostId

An integer value which identifies the virtual host. This parameter is reserved for future use and must always have a value of zero.

lpszUserName

A pointer to a string which specifies the user that will be removed. This parameter cannot be a NULL pointer or an empty string.

Return Value

If the method succeeds, the return value is non-zero. If the virtual host ID does not specify a valid host, the method will return zero. If the method fails, the last error code will be updated to indicate the cause of the failure.

Remarks

This method removes a virtual user that was created by a previous call to the **AddVirtualUser** method. This method will not match partial usernames and wildcard characters cannot be used to delete multiple users. Usernames are not case sensitive. You cannot use this method to delete the "anonymous" user.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AddVirtualUser](#)

CFtpServer::DetachHandle Method

```
HSERVER DetachHandle();
```

The **DetachHandle** method detaches the server handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the server handle associated with the current instance of the class object. If there is no active server, the value `INVALID_SERVER` will be returned.

Remarks

This method is used to detach a server handle created by the class for use with the SocketTools API. Once the server handle is detached from the class, no other class member functions may be called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftsv10.lib`

See Also

[AttachHandle](#), [GetHandle](#)

CFtpServer::DisableCommand Method

```
BOOL DisableCommand(  
    LPCTSTR lpszCommand  
);
```

Disable a specific server command.

Parameters

lpszCommand

A pointer to a NULL terminated string that specifies the name of the command to be enabled or disabled. The command name is not case-sensitive, but the value must otherwise match the exact name. Partial matches are not recognized by this method. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the command is not recognized, the method will return zero. If the method fails, the **GetLastError** method will return more information about the last error that has occurred.

Remarks

The **DisableCommand** method is used to disable access to a specific command on the server. When a command is disabled, it will also disable any corresponding feature related to that command. For example, if the MDTM command is disabled and a client issues the FEAT command to request a list of supported features, the command will no longer be listed. This method is typically used to disable certain commands for compatibility with older client software. The **IsCommandEnabled** method can be used to determine if a command is enabled or not.

The command name provided to this method must match the commands defined in RFC 959 or related protocol standards. It is important to distinguish between commands recognized by an FTP server and the commands that client programs may use. For example, the standard Windows FTP command line program provides commands such as GET and PUT to download and upload files. However, those are not the actual commands sent to a server. Instead, the corresponding server commands issued by a client application would be RETR (retrieve) and STOR (store). Refer to [File Transfer Protocol Commands](#) for a complete list of server commands.

Some commands cannot be disabled because they are required to perform essential server functions. For example, the USER and PASS commands are required to perform client authentication and therefore cannot be disabled. If you attempt to disable a required command, this method will return zero and the last error code will be set to ST_ERROR_COMMAND_REQUIRED. Because this method affects all clients connected to the server, it should not be used to limit access to certain commands for specific clients. Instead, either assign the client the appropriate permissions using the **AuthenticateClient** method, or use an event handler to filter the commands.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AuthenticateClient](#), [EnableCommand](#), [IsCommandEnabled](#)

CFtpServer::DisableTrace Method

```
BOOL DisableTrace();
```

Disable the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[EnableTrace](#)

CFtpServer::DisconnectClient Method

```
BOOL DisconnectClient(  
    UINT nClientId  
);
```

Close the control connection for the specified client and release the resources allocated for the session

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **DisconnectClient** method will close the control channel, disconnecting the client from the server and terminating the client session thread. Resources that we allocated for the client, such as memory and open handles, will be released back to the operating system. If the client was in the process of transferring a file, the transfer will be aborted. This performs the same operation as if the client sent the QUIT command to the server.

This method sends an internal control message that notifies the server that this session should be terminated. When the session thread is signaled that it should terminate, it will abort any active file transfers and begin to release the resources allocated for that session. To ensure that the client session terminates gracefully, there may be a brief period of time where the session thread is still active after this method has returned.

After this method returns, the application should never use the same client ID with another method. Client IDs are unique to the session over the lifetime of the server, and are not reused.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[Restart](#), [Stop](#)

CFtpServer::EnableClientAccess Method

```
BOOL EnableClientAccess(  
    UINT nClientId,  
    DWORD dwUserAccess,  
    BOOL bEnable  
);
```

Enable or disable access rights for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies an access right to enable or disable. For a list of user access rights that can be granted to the client, see [User Access Constants](#).

bEnable

An integer value which specifies if permission should be granted or revoked for the specified access right. If this value is non-zero, permission is granted to the client to perform the action specified by the *dwUserAccess* parameter. If this value is zero, that permission is revoked.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero. This method can only be used with authenticated clients. If the client session has not been authenticated, the return value will be zero.

Remarks

The **EnableClientAccess** method is used to enable or disable access to specific functionality by the client. The method can only change a single access right and cannot be used to enable or disable multiple access rights in a single method call. To change multiple user access rights for the client, use the **SetClientAccess** method.

This method cannot be used to change the access rights for a restricted or anonymous user. Those rights are granted when the client session is authenticated and will persist until the client disconnects from the server. This restriction is designed to prevent the inadvertent granting of rights to an untrusted client that could compromise the security of the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftsv10.lib

See Also

[AuthenticateClient](#), [GetClientAccess](#), [SetClientAccess](#)

CFtpServer::EnableCommand Method

```
BOOL EnableCommand(  
    LPCTSTR LpszCommand  
);
```

Enable a specific server command.

Parameters

LpszCommand

A pointer to a NULL terminated string that specifies the name of the command to be enabled or disabled. The command name is not case-sensitive, but the value must otherwise match the exact name. Partial matches are not recognized by this method. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the command is not recognized, the method will return zero. If the method fails, the **GetLastError** method will return more information about the last error that has occurred.

Remarks

The **EnableCommand** method is used to enable access to a specific command on the server. When a command is enabled, it will also enable any corresponding feature related to that command. For example, if the MDTM command is enabled and a client issues the FEAT command to request a list of supported features, the command will be included in the list. The **IsCommandEnabled** method can be used to determine if a command is enabled or not.

The command name provided to this method must match the commands defined in RFC 959 or related protocol standards. It is important to distinguish between commands recognized by an FTP server and the commands that client programs may use. For example, the standard Windows FTP command line program provides commands such as GET and PUT to download and upload files. However, those are not the actual commands sent to a server. Instead, the corresponding server commands issued by a client application would be RETR (retrieve) and STOR (store). Refer to [File Transfer Protocol Commands](#) for a complete list of server commands.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AuthenticateClient](#), [DisableCommand](#), [IsCommandEnabled](#)

CFtpServer::EnableTrace Method

```
BOOL EnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

Enable the logging of network function calls to a file.

Parameters

lpszTraceFile

A pointer to a string that specifies the name of the log file. If this parameter is NULL or points to an empty string, a log file is created in the temporary directory for the current user.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_DEFAULT (0)	All function calls are written to the trace file. This is the default value.
TRACE_ERROR (1)	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING (2)	Only those function calls which fail, or return values which indicate a warning, are recorded in the trace file.
TRACE_HEXDUMP (4)	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

When trace logging is enabled, the log file is opened, appended to and closed for each socket function call. Using the same file name, you can do the same in your application to add additional information to the file if needed. This can provide an application-level context for the entries made by the library. Make sure that the file is closed after the data has been written. If a file name is not specified by the caller, a file named **cstrace.log** will be created in the temporary directory for the current user.

The TRACE_HEXDUMP option can produce very large files, since all data that is being sent and received by the application is logged. To reduce the size of the file, you can enable and disable logging around limited sections of code that you wish to analyze.

To redistribute an application that includes this debug logging functionality, the **cstrcv10.dll** library must be included as part of the installation package. This library provides the trace logging features, and if it is not available the **EnableTrace** method will fail. Note that this is a standard Windows DLL and does not need to be registered, it only needs to be redistributed with your application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableTrace](#)

CFtpServer::EnumClients Method

```
INT EnumClients(  
    UINT * lpClients,  
    INT nMaxClients  
);
```

Return a list of active client connections established with the server.

Parameters

lpClients

Pointer to an array of unsigned integers which will contain client IDs that uniquely identifies each client when the method returns. If this parameter is NULL, then the method will return the number of active client connections established with the server.

nMaxClients

Maximum number of client IDs to be returned in the *lpClients* array. If the *lpClients* parameter is NULL, this parameter should have a value of zero.

Return Value

If the method succeeds, the return value is the number of active client connections to the server. If the method fails, the return value is FTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the *nMaxClients* parameter is less than the number of active client connections, the method will fail and the last error code will be set to the error ST_ERROR_BUFFER_TOO_SMALL. To dynamically determine the number of active connections, call the method with the *lpClients* parameter with a value of NULL, and the *nMaxClients* parameter with a value of zero.

Example

```
// Populate a listbox with all of the users connected to the server  
pListBox->ResetContent();  
  
INT nClients = pFtpServer->EnumClients();  
if (nClients > 0)  
{  
    UINT *pIdList = new UINT[nClients];  
  
    nClients = pFtpServer->EnumClients(pIdList, nClients);  
    if (nClients == FTP_ERROR)  
    {  
        // Unable to obtain list of connected clients  
        return;  
    }  
  
    for (INT nIndex = 0; nIndex < nClients; nIndex++)  
    {  
        CString strUserName;  
  
        if (pFtpServer->GetClientUserName(pIdList[nIndex], strUserName))  
            pListBox->AddString(strUserName);  
    }  
  
    // Free the memory allocated for the client IDs
```

```
    delete pIdList;  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[GetClientAddress](#), [GetClientDirectory](#), [GetClientUserName](#)

CFtpServer::GetActiveClient Method

```
UINT GetActiveClient();
```

Return the client ID for the active client session associated with the current thread.

Parameters

None.

Return Value

If the method succeeds, the return value is the unique ID associated with the client session for the current thread. If there is no client session active on the current thread, the return value is zero.

Remarks

The **GetActiveClient** method is used to obtain the client ID associated with the current thread. This means this method will only return a client ID if it is called within an event handler or a method called by an event handler. If this method is called by a function that is not executing within the context of an event handler it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[EnumClients](#)

CFtpServer::GetAddress Method

```
INT GetAddress(  
    UINT nClientId,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

```
INT GetAddress(  
    UINT nClientId,  
    CString& strAddress  
);
```

Return the IP address of the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszAddress

A pointer to a string buffer that will contain the server IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the client ID is invalid, or the buffer is not large enough to store the complete address, the method will return a value of zero.

Remarks

This method will return the IP address assigned to the specified server as a printable string. If the *nClientId* parameter has a value of zero, this method will return the IP address assigned to the local system. If the FTP_SERVER_NATROUTER option was specified when the server was started, this method will return the external IP address assigned to the system. If the *nClientId* parameter specifies a valid client session, this method will return the IP address that the client used to establish the connection with the server. To determine the IP address assigned to the client, use the **GetClientAddress** method.

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientAddress](#), [GetName](#), [SetAddress](#)

CFtpServer::GetClientAccess Method

```
BOOL GetClientAccess(  
    UINT nClientId,  
    DWORD& dwUserAccess  
);
```

Return the access rights that have been granted to the client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies one or more access rights for the client session. For a list of user access rights that can be granted to the client, see [User Access Constants](#).

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero. This method can only be used with authenticated clients. If the client session has not been authenticated, the return value will be zero.

Remarks

The **GetClientAccess** method is used to obtain all of the access rights that are currently granted to an authenticated client session. The **EnableClientAccess** method can be used to enable or disable specific permissions, and the **SetClientAccess** method can change multiple access rights at once.

Example

```
DWORD dwUserAccess = 0;  
  
// Check if the client is a restricted user  
if (pFtpServer->GetClientAccess(nClientId, dwUserAccess))  
{  
    if (dwUserAccess & FTP_ACCESS_RESTRICTED)  
    {  
        std::cout << "Client authenticated as a restricted user\n";  
        return;  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[AuthenticateClient](#), [EnableClientAccess](#), [SetClientAccess](#)

CFtpServer::GetClientAddress Method

```
INT GetClientAddress(  
    UINT nClientId,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

```
INT GetClientAddress(  
    UINT nClientId,  
    CString& strAddress  
);
```

Return the IP address of the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszAddress

A pointer to a string buffer that will contain the client IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID is invalid, or the buffer is not large enough to store the complete address, the method will return a value of zero.

Remarks

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientDirectory](#), [GetClientUserName](#)

CFtpServer::GetClientCredentials Method

```
BOOL GetClientCredentials(  
    UINT nClientId,  
    LPFTPCLIENTCREDENTIALS lpCredentials  
);  
  
BOOL GetClientCredentials(  
    UINT nClientId,  
    CString& strUserName,  
    CString& strPassword  
);
```

Return the user credentials for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpCredentials

A pointer to an **FTPCLIENTCREDENTIALS** structure that will contain information about the user when the method returns. This parameter cannot be NULL.

strUserName

A string that will contain the user name when the method returns. This version of the method is only available for MFC and ATL based projects that define the **CString** object.

strPassword

A string that will contain the user password when the method returns. This version of the method is only available for MFC and ATL based projects that define the **CString** object.

Return Value

If the user credentials for the client session are available, the return value is non-zero. If the client ID does not specify a valid client session, or the client has not requested authentication, this method will return zero.

Remarks

The **GetClientCredentials** method is used to obtain the username and password that was provided by the client when it requested authentication. Typically this method is used in an event handler to validate the credentials provided by the client. If the credentials are considered valid, the event handler would then call the **AuthenticateClient** method to specify that the session has been authenticated.

If the default event handler is used, the **OnAuthenticate** method will be invoked with the user credentials passed to the handler as arguments.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

CFtpServer::GetClientDirectory Method

```
INT GetClientDirectory(  
    UINT nClientId,  
    LPTSTR lpszDirectory,  
    INT nMaxLength  
);  
  
INT GetClientDirectory(  
    UINT nClientId,  
    CString& strDirectory  
);
```

Returns the current working directory for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszDirectory

A pointer to a string buffer that will contain the current working directory for the specified client session, terminated with a null character. This buffer should be at least MAX_PATH characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be larger than zero or the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID does not specify a valid client session, the method will return zero.

Remarks

This method returns the full path to the current working directory for the specified client session. For example, if the server root directory is C:\ProgramData\MyServer and the current working directory for the client is /Research/Documents, this method will return C:\ProgramData\MyServer\Research\Documents as the current working directory for the client session.

It is important to note that the current working directory for client sessions is virtual, and does not reflect the current working directory for the server process. To change the current working directory for a client, use the **ChangeClientDirectory** method.

This method should only be used with client sessions that have been authenticated. Unauthenticated clients are not assigned a current working directory and this method will return zero, with the last error code set to ST_ERROR_AUTHENTICATION_REQUIRED.

To convert a full path to the virtual path for a specific client session, use the **GetClientVirtualPath** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ChangeClientDirectory](#), [GetClientHomeDirectory](#), [GetClientVirtualPath](#),

CFtpServer::GetClientFileType Method

```
BOOL GetClientFileType(  
    UINT nClientId,  
    UINT& nFileType  
);
```

Return the current file type used for transfers by the specified client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpnFileType

An unsigned integer value that will contain the current file type used by the client for data transfers.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **GetClientFileType** method will return the current file type that has been specified by the client sending the TYPE command to the server. The file type determines if there is any conversion performed on the data that is being exchanged between the client and server. The following file types are supported:

Value	Description
FILE_TYPE_ASCII (1)	The file is a text file using the ASCII character set. For those clients which use a different end-of-line character sequence, the text file has been converted to the local format which uses the carriage return (CR) and linefeed (LF) characters.
FILE_TYPE_IMAGE (3)	The file is a binary file and no data conversion of any type has been performed on the file. This is the default file type for most data files and executable programs. If the client specified this file type when appending to a text file, the file will contain the end-of-line sequences used by its native operating system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[SetClientFileType](#)

CFtpServer::GetClientHomeDirectory Method

```
INT GetClientHomeDirectory(  
    UINT nClientId,  
    LPTSTR lpszDirectory,  
    INT nMaxLength  
);
```

```
INT GetClientHomeDirectory(  
    UINT nClientId,  
    CString& strDirectory  
);
```

Returns the home directory for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszDirectory

A pointer to a string buffer that will contain the home directory for the specified client session, terminated with a null character. This buffer should be at least MAX_PATH characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be larger than zero or the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID does not specify a valid client session, the method will return zero.

Remarks

This method returns the full path to the home working directory assigned to the specified client session. This will be the same path to the home directory specified when the **AuthenticateClient** method was used to authenticate the client session. If a home directory was not explicitly assigned when the client was authenticated, then this method returns the default home directory that was created for the client, or the server root directory if the FTP_SERVER_MULTUSER option was not specified when the server was started.

This method should only be used with client sessions that have been authenticated. Unauthenticated clients are not assigned a home directory and this method will return zero, with the last error code set to ST_ERROR_AUTHENTICATION_REQUIRED.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AuthenticateClient](#), [ChangeClientDirectory](#), [GetClientDirectory](#), [GetClientVirtualPath](#)

CFtpServer::GetClientIdentity Method

```
INT GetClientIdentity(  
    UINT nClientId,  
    LPTSTR lpszIdentity,  
    INT nMaxLength  
);
```

```
INT GetClientIdentity(  
    UINT nClientId,  
    CString& strIdentity  
);
```

Return the identity of the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszIdentity

A pointer to a string buffer that will contain the identity of the client when the method returns, terminated with a null character. This parameter cannot be NULL. It is recommended that this buffer be at least 32 characters in length. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID is invalid, or the buffer is not large enough to store the complete path, the method will return a value of zero. If the client did not identify itself, this method will return zero.

Remarks

The **GetClientIdentity** method returns the string that the client used to identify itself to the server. The client may use either the CLNT or CSID command to identify itself. Although the CLNT command is considered to be deprecated, it is supported for backwards compatibility with older clients. The identity string does not have any standard format and is used for informational purposes only and does not affect the operation of the server in any way. Not all clients identify themselves, in which case this method will return zero and the *lpszIdentity* string buffer will be set to an empty string.

If the client does identify itself, it typically uses the name of the client application that was used to establish the connection. The application may choose to assign an identity to a client session for its own internal purposes using the **SetClientIdentity** method, regardless of whether the client identifies itself.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetIdentity](#), [SetClientIdentity](#), [SetIdentity](#)

CFtpServer::GetClientIdleTime Method

```
UINT GetClientIdleTime(  
    UINT nClientId,  
    UINT * lpnElapsed  
);
```

Return the idle timeout period for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpnElapsed

An optional pointer to an unsigned integer value that will contain the number of seconds the client session has been idle. This parameter may be NULL or omitted if this information is not required.

Return Value

If the method succeeds, the return value is client idle timeout period in seconds. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **GetClientIdleTime** method will return the number of seconds that the client may remain idle before being automatically disconnected by the server. The idle time of a client session is based on the last time a command was issued to the server or when a file data transfer completed. The server will never disconnect a client that is in the process of uploading or downloading a file, regardless of the idle timeout period.

The default idle timeout period for a client is 900 seconds (15 minutes), however the server can be configured to use a different value and individual clients can request the timeout period be changed by sending the IDLE command to the server. For a client to be able to change its own timeout period, it must be granted the FTP_ACCESS_IDLE permission. The minimum timeout period for a client is 60 seconds, the maximum is 7200 seconds (2 hours). An application can change the timeout period for a specific client session using the **SetClientIdleTime** method.

It is important to note that the idle timeout period only affects authenticated clients.

Unauthenticated clients use a different internal timer that limits the amount of time they can remain connected to the server before successfully authenticating with a valid username and password. By default, the authentication timeout period is 60 seconds and is set when the server is started; it cannot be changed for an individual client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[SetClientIdleTime](#)

CFtpServer::GetClientLocalPath Method

```
INT GetClientLocalPath(  
    UINT nClientId,  
    LPCTSTR lpszVirtualPath,  
    LPTSTR lpszLocalPath,  
    INT nMaxLength,  
);
```

```
INT GetClientLocalPath(  
    UINT nClientId,  
    LPCTSTR lpszVirtualPath,  
    CString& strLocalPath  
);
```

Return the full local path for a virtual filename or directory on the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszVirtualPath

A pointer to a string that specifies an virtual path on the server. This parameter cannot be NULL.

lpszLocalPath

A pointer to a string buffer that will contain the full local path, terminated with a null-character. This buffer should be at least MAX_PATH characters to accommodate the complete path. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID does not specify a valid client session, the method will return zero. If the string buffer is not large enough to contain the complete path, this method will return zero and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL.

Remarks

The **GetClientLocalPath** method takes a virtual path and returns the full path to the specified file or directory on the local system. The virtual path may be absolute or relative to the current working directory for the client session. This method will recognize a tilde at the beginning of the path to specify the client home directory.

To obtain the virtual path for a local file or directory, use the **GetClientVirtualPath** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientVirtualPath](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetClientServer Method

```
HSERVER HttpGetClientServer(  
    UINT nClientId  
);
```

The **GetClientServer** method returns a handle to the server that created the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the method succeeds, the return value is the handle to the server that created the client session. If the method fails, the return value is `INVALID_SERVER`. To get extended error information, call the **GetLastError** method.

Remarks

The **GetClientServer** method returns the handle to the server that created the client session and is typically used within a notification message handler. If the server is in the process of shutting down, or the client session thread is terminating, this method will fail and return `INVALID_SERVER` indicating that the session ID is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `csftools10.h`.

Import Library: `csftsv10.lib`

See Also

[AsyncNotify](#)

CFtpServer::GetClientThreadId Method

```
DWORD GetClientThreadId(  
    UINT nClientId  
);
```

Returns the thread ID associated with the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the method succeeds, the return value is a thread ID. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **GetClientThreadId** method returns a thread ID that can be used to identify the thread that is managing the client session. The thread ID can be used with other Windows API functions such as **OpenThread**. Exercise caution when using thread-related functions, interfering with the normal operation of the thread can have unexpected results. You should never use this method to obtain a thread handle and then call the **TerminateThread** function to terminate a client session. This will prevent the thread from releasing the resources that were allocated for the session and can leave the server in an unstable state. To terminate a client session, use the **DisconnectClient** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[EnumClients](#), [GetActiveClient](#)

CFtpServer::GetClientUserName Method

```
INT GetClientUserName(  
    UINT nClientId,  
    LPTSTR lpszUserName,  
    INT nMaxLength  
);
```

```
INT GetClientUserName(  
    UINT nClientId,  
    CString& strUserName  
);
```

Return the user name associated with the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

strUserName

A pointer to a string buffer that will contain the user name associated with the client session. This buffer must be large enough to store the complete user name, including the terminating null character. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, or the client has not authenticated itself, the method will return zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AuthenticateClient](#), [GetClientAccess](#), [GetClientHomeDirectory](#)

CFtpServer::GetClientVirtualPath Method

```
INT GetClientVirtualPath(  
    UINT nClientId,  
    LPCTSTR lpszLocalPath,  
    LPTSTR lpszVirtualPath,  
    INT nMaxLength,  
);
```

```
INT GetClientVirtualPath(  
    UINT nClientId,  
    LPCTSTR lpszLocalPath,  
    CString& strVirtualPath  
);
```

Return the virtual path for a local file on the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszLocalPath

A pointer to a string that specifies an absolute path on the local system. This parameter cannot be NULL.

lpszVirtualPath

A pointer to a string buffer that will contain the virtual path, terminated with a null-character. This buffer should be at least MAX_PATH characters to accommodate the complete path. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID does not specify a valid client session, the method will return zero. If the string buffer is not large enough to contain the complete path, this method will return zero and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL.

Remarks

A virtual path for the client is either relative to the server root directory, or the client home directory if the client was authenticated as a restricted user. These virtual paths are what the client will see as an absolute path on the server. For example, if the server was configured to use "C:\ProgramData\MyServer" as the root directory, and the *lpszLocalPath* parameter was specified as "C:\ProgramData\MyServer\Documents\Research", this method would return the virtual path to that directory as "/Documents/Research".

If the client session was authenticated as a restricted user, then the virtual path is always relative to the client home directory instead of the server root directory. This is because restricted users are isolated to their own home directory and any subdirectories. For example, if restricted user "John" has a home directory of "C:\ProgramData\MyServer\Users\John" and the *lpszLocalPath* parameter was specified as "C:\ProgramData\MyServer\Users\John\Accounting\Projections.pdf"

this method would return the virtual path as `"/Accounting/Projections.pdf"`.

If the *lpzLocalPath* parameter specifies a file or directory outside of the server root directory, this method will return zero and the last error code will be set to `ST_ERROR_INVALID_FILE_NAME`. This method can only be used with authenticated clients. If the *nClientId* parameter specifies a client session that has not been authenticated, this method will return zero and the last error code will be `ST_ERROR_AUTHENTICATION_REQUIRED`.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetCommandFile](#), [GetClientLocalPath](#)

CFtpServer::GetCommandFile Method

```
INT GetCommandFile(  
    UINT nClientId,  
    LPTSTR lpszFileName,  
    INT nMaxLength  
);  
  
INT GetCommandFile(  
    UINT nClientId,  
    CString& strFileName  
);
```

Get the full path to the local file name or directory specified by the client

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A pointer to a string buffer that will contain the full path to a file name or directory specified by the client when it issued a command. The string buffer will be null terminated and must be large enough to store the complete file path. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. It is recommended that the buffer be at least MAX_PATH characters in size. If the maximum length specified is smaller than the actual length of the full path, this method will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code. If the last error code is returned as a value of zero, this means that the command issued by the client accepts a file name as an argument, but the client did not specify one.

Remarks

The **GetCommandFile** method is used to obtain the full path to a local file name or directory specified by the client as an argument to a standard FTP command. For example, if the client sends the RETR command to the server, this method will return the complete path to the local file that the client wants to download. This method will only work with those standard commands that perform some action on a file or directory.

This method should always be used to obtain the file name for a command that performs a file or directory operation. The **GetCommandParam** method will return the actual command parameter, but the file name will typically be relative to the user home directory or server root directory, and cannot be passed directly to a Windows API function. The **GetCommandFile** method normalizes the path provided by the client and ensures that it specifies a file or directory name in the correct location.

To change the file or directory name that is the target of the current command, use the **SetCommandFile** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientDirectory](#), [GetClientHomeDirectory](#), [GetCommandLine](#), [GetCommandParam](#),
[SetCommandFile](#)

CFtpServer::GetCommandLine Method

```
INT GetCommandLine(  
    UINT nClientId,  
    LPTSTR lpszCmdLine,  
    INT nMaxLength  
);
```

```
INT GetCommandLine(  
    UINT nClientId,  
    CString& strCmdLine  
);
```

Return the complete command line issued by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCmdLine

A pointer to a string buffer that will contain the command, including all arguments. The string buffer will be null terminated and must be large enough to store the complete command line. If this parameter is NULL, the method will return the length of the command line. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. The internal limit on the maximum length of a command is 1024 characters. If the maximum length specified is smaller than the actual length of the complete command, this method will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code. If the last error code has a value of zero then no command has been issued by the client.

Remarks

The **GetCommandLine** method is used to obtain the command that was issued by the client, and is commonly used inside the **OnCommand** and **OnResult** event handlers to pre-process and post-process client commands, respectively. When the method returns, the string buffer provided by the caller will contain the complete command, including all command parameters. Any extraneous whitespace will be removed, however quoted parameters will be retained as-is.

To obtain a specific parameter to a command, use the **GetCommandParam** method. The **GetCommandParamCount** method will return the number of command parameters that were provided by the client. If the command sent by the client is used to perform an action on a file or directory, the **GetCommandFile** method should be called to obtain the full path to the specified file rather than using the value of the command parameter.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetCommandFile](#), [GetCommandParam](#), [OnCommand](#), [OnResult](#)

CFtpServer::GetCommandName Method

```
INT GetCommandName
    UINT nClientId,
    LPTSTR lpszCommand,
    INT nMaxLength
);
```

```
INT GetCommandName
    UINT nClientId,
    CString& strCommand
);
```

Return the name of the last command issued by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCommand

A pointer to a string buffer that will contain the command name. The string buffer will be null terminated and must be large enough to store the complete command name. If this parameter is NULL, the method will only return the length of the current command in characters, not including the terminating null character. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. If the maximum length specified is smaller than the actual length of the parameter, this method will fail. If the *lpszCommand* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code. If the last error code is returned as a value of zero, this means that no command has been issued by the client.

Remarks

The **GetCommandName** method is used to obtain the name of the last command that was issued by the client. The command name returned by this method will always be capitalized, regardless of how it was sent by the client. This method is typically used inside the **OnCommand** and **OnResult** event handlers to pre-process and post-process client commands, respectively.

The **GetCommandParam** method can be used to return the value of individual command parameters specified by the client. The **GetComandLine** method can be used to obtain the complete command line issued by the client.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetCommandFile](#), [GetCommandLine](#), [GetCommandParam](#), [OnCommand](#), [OnResult](#)

CFtpServer::GetCommandParam Method

```
INT GetCommandParam
    UINT nClientId,
    INT nParam,
    LPTSTR lpszParam,
    INT nMaxLength
);
```

```
INT GetCommandParam
    UINT nClientId,
    INT nParam,
    CString& strParam
);
```

Return the value of the specified command parameter from the last command issued by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nParam

An integer value which specifies the command parameter. A value of zero specifies the command itself, while values greater than zero specify a particular parameter. This method will fail if this value is less than zero or greater than the number of parameters available.

lpszParam

A pointer to a string buffer that will contain the command parameter. The string buffer will be null terminated and must be large enough to store the complete parameter value. If this parameter is NULL, the method will only return the length of the specified parameter in characters, not including the terminating null character. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. The internal limit on the maximum length of a command is 1024 characters. If the maximum length specified is smaller than the actual length of the parameter, this method will fail. If the *lpszParam* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code. If the last error code is returned as a value of zero, this means that no command has been issued by the client.

Remarks

The **GetCommandParam** method is used to obtain a specific parameter for the last command that was issued by the client. If the parameter was surrounded in quotes, those quotes will be included in the value returned by this method. This method is typically used inside the **OnCommand** and **OnResult** event handlers to pre-process and post-process client commands, respectively.

The **GetCommandParamCount** method will return the number of command parameters that

were provided by the client. If the command sent by the client is used to perform an action on a file or directory, the **GetCommandFile** method should be called to obtain the full path to the specified file rather than using the value of the command parameter. To obtain the complete command line, use the **GetCommandLine** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetCommandFile](#), [GetCommandLine](#), [GetCommandName](#), [GetCommandParamCount](#), [OnCommand](#), [OnResult](#)

CFtpServer::GetCommandParamCount Method

```
INT GetCommandParamCount  
    UINT nClientId  
);
```

Return the number of command parameters for the last command issued by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

An integer value which specifies the number of parameters that were specified in the last command issued by the client. If the command did not include any parameters, this method will return zero. If the client has not issued a command, or the client session ID is invalid, this method will return -1.

Remarks

The **GetCommandParamCount** method is used to determine the number of parameters specified in the last client command, and the maximum value that may be passed as the parameter index to the **GetCommandParam** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[GetCommandLine](#), [GetCommandName](#), [GetCommandParam](#)

CFtpServer::GetCommandResult Method

```
INT GetCommandResult(  
    UINT nClientId,  
    LPTSTR lpszResult,  
    INT nMaxLength  
);  
  
INT GetCommandResult(  
    UINT nClientId,  
    CString& strResult  
);
```

Return the result code and description for the last command issued by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszResult

A pointer to a string buffer that will contain the description of the result code. The string buffer will be null terminated up to the maximum number of characters specified by the caller. This parameter can be NULL if this information is not required. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. If the *lpszResult* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the result code for the last command issued by the client. A return value of zero indicates that the command has not completed and there is no result code available.

Remarks

The **GetCommandResult** method is used to determine the result of the last command that was issued by the client and is typically called in the **OnResult** event handler. This method should only be called after a command has been processed or the **SendResponse** method has been called.

The result code is a three-digit integer value that indicates the success or failure of a command. Whenever a client sends a command to the server, the server must respond with this numeric code, and optionally a text message that further describes the result. The text message may be a single line, or it may span multiple lines, with each line of text terminated by a carriage return and linefeed. Result codes are generally broken down into the following categories:

Result Code	Description
100-199	Result codes in this range indicate that the requested action is being initiated, and the client should expect another reply from the server before proceeding. This is normally used with file transfers, indicating to the client that the data transfer has started.
200-299	Result codes in this range indicate that the server has successfully completed the requested action. One exception is the 202 result code which indicates that the command is not implemented, but the client should not consider this to be an error condition.

300-399	Result codes in this range indicate that the requested action cannot complete until additional information is provided to the server. This is normally used with commands that require a specific sequence to complete. For example, the server will send the 331 result code in response to the USER command, which tells the client that it must send the PASS command to complete the authentication process.
400-499	Result codes in this range indicate that the requested action did not take place, but the error condition is temporary and may be attempted again. This error response is usually the result of a failed authentication attempt or a file transfer that could not complete.
500-599	Result codes in this range indicate that the requested action did not take place and the failure is permanent. The client should not attempt to send the command again. This error response is usually the result of an invalid command name, a syntax error or the client not having the appropriate access rights to a resource.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SendResponse](#), [OnResult](#)

CFtpServer::GetCommandUsage Method

```
UINT GetCommandUsage(  
    LPCTSTR lpszCommand  
);
```

Return the number of times a specific command has been issued by all clients.

Parameters

lpszCommand

A pointer to a string that specifies a command name. The name is not case-sensitive, but must match a valid server command exactly. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is the number of times the command has been issued by all clients since the server was started. If the command name is invalid, or the command has never been issued, the return value will be zero.

Remarks

The command name provided to this method must match the commands defined in RFC 959 or related protocol standards. It is important to distinguish between commands recognized by an FTP server and the commands that client programs may use. For example, the standard Windows FTP command line program provides commands such as GET and PUT to download and upload files. However, those are not the actual commands sent to a server. Instead, the corresponding server commands issued by a client application would be RETR (retrieve) and STOR (store). Refer to [File Transfer Protocol Commands](#) for a complete list of server commands.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[EnableCommand](#), [GetCommandLine](#), [GetCommandResult](#)

CFtpServer::GetDirectory Method

```
INT GetDirectory(  
    LPTSTR lpszDirectory,  
    INT nMaxLength  
);  
  
INT GetDirectory(  
    CString& strDirectory  
);
```

Return the full path to the root directory assigned to the specified server.

Parameters

lpszDirectory

A pointer to a string buffer that will contain the server root directory, terminated with a null character. It is recommended that this buffer be at least MAX_PATH characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the buffer is not large enough to store the complete path, the method will return a value of zero.

Remarks

The **GetDirectory** method will return the full path to the root directory assigned to the server instance. The root directory may be specified as part of the server configuration, or if no directory is specified by the application, the current working directory will be used and this method can be used to obtain the full path to the directory. When the application specifies a root directory, it may use environment variables such as %AppData% in the path. This method will return the fully resolved path name, with all environment variables expanded.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetIdentity](#), [GetName](#), [SetDirectory](#)

CFtpServer::GetHandle Method

```
HSERVER GetHandle();
```

The **GetHandle** method returns the server handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the server handle associated with the current instance of the class object. If the server is inactive, the value `INVALID_SERVER` will be returned.

Remarks

This method is used to obtain the server handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

See Also

[IsInitialized](#)

CFtpServer::GetIdentity Method

```
INT GetIdentity(  
    LPTSTR lpszIdentity,  
    INT nMaxLength  
);  
  
INT GetIdentity(  
    CString& strIdentity  
);
```

Return the identity of the specified server.

Parameters

lpszIdentity

A pointer to a string buffer that will contain the identity of the server when the method returns, terminated with a null character. This parameter cannot be NULL. It is recommended that this buffer be at least 32 characters in length. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the buffer is not large enough to store the complete path, the method will return a value of zero.

Remarks

The **GetIdentity** method returns the identity string that was specified as part of the server configuration. It is used for informational purposes only and does not affect the operation of the server. Typically the string specifies the name of the application and a version number, and is displayed whenever a client establishes its initial connection to the server. The **SetIdentity** method can be used to change the identity string associated with the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientIdentity](#), [SetClientIdentity](#), [SetIdentity](#)

CFtpServer::GetLastError Method

```
DWORD GetLastError(  
    LPTSTR lpszError,  
    INT nMaxLength  
);  
  
DWORD GetLastError(  
    CString& strError  
);  
  
DWORD GetLastError();
```

Return the last server error code and a description of the error.

Parameters

lpszError

A pointer to a string buffer that will contain a description of the error. If the error description is not needed, this parameter may be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the error string buffer, including the terminating null character. If the *lpszError* parameter is NULL, this value should be zero.

Return Value

An unsigned integer value that specifies the last error that occurred. A value of zero indicates that there was no error.

Remarks

Error codes are unsigned 32-bit values which are private to each server. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods clear the last error code when they succeed.

It is important to note that the error codes returned by this method are different than the command result codes that are defined in RFC 959, the standard protocol specification for FTP. This method is used to determine reason that an API function has failed, and should not be used to determine if a command issued by the client was successful. The **SendResponse** method is used to send result codes to the client, and the **GetCommandResult** method can be used to determine the result of the last command sent by the client.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Failure is typically indicated by a return value such as FALSE, NULL, INVALID_SERVER or FTP_ERROR. Those methods which clear the last error code when they succeed are noted on their reference page.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetCommandResult](#), [SendResponse](#), [SetLastError](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetLogFile Method

```
BOOL GetLogFile(  
    UINT * lpnLogFormat,  
    UINT * lpnLogLevel,  
    LPTSTR lpszFileName,  
    INT nMaxLength  
);
```

```
BOOL GetLogFile(  
    UINT * lpnLogFormat,  
    UINT * lpnLogLevel,  
    CString& strFileName  
);
```

```
BOOL GetLogFile(  
    CString& strFileName  
);
```

Return the current log file format and the full path to the file.

Parameters

lpnLogFormat

A pointer to an integer value that will contain the log file format being used when the method returns. If this information is not needed, this parameter may be NULL. The following formats are supported:

Constant	Description
FTP_LOGFILE_NONE (0)	This value specifies that the server should not create or update a log file.
FTP_LOGFILE_COMMON (1)	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about file transfers.
FTP_LOGFILE_EXTENDED (2)	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session.

lpnLogLevel

A pointer to an integer value that will contain the level of detail the server uses when generating the log file. The minimum value is 1 and the maximum value is 10. If this information is not needed, this parameter may be NULL.

lpszFileName

A pointer to a string buffer that will contain the full path to the log file. This parameter may be NULL if this information is not required. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the file name string, including the terminating null character. If the *lpszFileName* parameter is NULL,

this value should be zero.

Return Value

An integer value which specifies the current log file format. Refer to the **FTPSEVERCONFIG** structure definition for a list of supported log file formats. If logging has not been enabled, this method will return a value of zero.

Remarks

If the server is configured with logging enabled, but a log file name is not explicitly provided, then the server will automatically generate one. This method can be used to get the full path to the current log file along with the format that is being used to record client session data. Normally the log file is held open by the server thread while it is active, however you can call the **RenameServerLogFile** method to explicitly rename or delete the log file.

To change the name of the log file, the log file format or level of detail, use the **SetLogFile** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RenameServerLogFile](#), [SetLogFile](#)

GetMemoryUsage Method

```
SIZE_T GetMemoryUsage();
```

Return the amount of memory allocated for the server and all client sessions.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero and specifies the amount of memory allocated by the server. If the server is inactive or cannot be locked, the return value is zero. Call the **GetLastError** method to determine the cause of the failure.

Remarks

This method returns the amount of memory allocated by the server and all active client sessions. It enumerates all of memory allocations made by the server process and client session threads and returns the total number of bytes allocated for the server process. This value reflects the amount of memory explicitly allocated by this library and does not reflect the total working set size of the process, or memory allocated by any other libraries. To determine the working set size for the process, refer to the Win32 **GetProcessWorkingSetSize** and **GetProcessMemoryInfo** functions.

This method forces the server into a locked state, and all client sessions will block until the method returns. Because this method enumerates all heaps allocated for the server process, it can be an expensive operation, particularly when there are a large number of active clients connected to the server. Frequent use of this method can significantly degrade the performance of the server. It is primarily intended for use as a debugging tool to determine if memory usage is the result of an increase in active client sessions. If the value returned by the method remains reasonably constant, but the amount of memory allocated for the process continues to grow, it could indicate a memory leak in some other area of the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetStackSize](#), [SetStackSize](#)

CFtpServer::GetName Method

```
INT GetName(  
    UINT nClientId,  
    LPTSTR lpszHostName,  
    INT nMaxLength  
);  
  
INT GetName(  
    UINT nClientId,  
    CString& strHostName  
);
```

Return the host name assigned to the specified server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszHostName

A pointer to a string buffer that will contain the server host name, terminated with a null character. It is recommended that this buffer be at least 64 characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID is invalid or the buffer is not large enough to store the complete hostname, the method will return a value of zero.

Remarks

This method will return the host name assigned to the specified server. If the *nClientId* parameter has a value of zero, the method will return the default host name that was specified as part of the server configuration. If no host name was explicitly assigned to the server, then it will return the local system name. If the *nClientId* parameter specifies a client session, then it this method will return the host name that the client used to establish the connection. If the client sends the HOST command to the server, this method will return the host name provided by the client.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetAddress](#)

CFtpServer::GetOptions Method

```
DWORD GetOptions();
```

Return the options specified for this instance of the server.

Parameters

None.

Return Value

The current server options. For a list of available options, see [Server Option Constants](#)

Remarks

The **GetOptions** method returns the default options for the current instance of the server. To change the server options, use the **SetOptions** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[SetOptions](#)

CFtpServer::GetPriority Method

```
INT GetPriority();
```

Return the current priority assigned to the specified server.

Parameters

None.

Return Value

If the method succeeds, the return value is the priority for the specified server. If the method fails, the return value is `FTP_PRIORITY_INVALID`. To get extended error information, call the **GetLastError** method.

Remarks

The **GetPriority** method can be used to determine the current priority assigned to the server. It will return one of the following values:

Constant	Description
FTP_PRIORITY_BACKGROUND (0)	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
FTP_PRIORITY_LOW (1)	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
FTP_PRIORITY_NORMAL (2)	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
FTP_PRIORITY_HIGH (3)	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
FTP_PRIORITY_CRITICAL (4)	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `csftools10.h`.

Import Library: `csftsv10.lib`

See Also

[SetPriority](#), [Start](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CFtpServer::GetProgramExitCode Method

```
BOOL GetProgramExitCode(  
    UINT nClientId,  
    DWORD& dwExitCode  
);
```

Return the exit code of the last program executed by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwExitCode

An unsigned integer that will contain the program exit code when the method returns.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **GetProgramExitCode** method returns the exit code of a registered program that was executed by the client using the SITE EXEC command. By convention, most programs return an exit code in the range of 0-255, with an exit code of zero indicating success. The exit code is commonly used by custom programs to communicate status information back to the server application.

Permission to use the SITE EXEC is not granted to authenticated users by default, and is limited to only those programs which are explicitly registered with the server. Exercise caution when allowing a client to execute a program on the server because this can expose the server to significant security risks. The programs that are registered for use with the SITE EXEC command should be thoroughly tested before being deployed on the server and should only be console programs that write to standard output.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[GetProgramName](#), [GetProgramOutput](#), [RegisterProgram](#)

CFtpServer::GetProgramName Method

```
INT GetProgramName(  
    UINT nClientId,  
    LPTSTR lpszProgramName,  
    INT nMaxLength  
);  
  
INT GetProgramName(  
    UINT nClientId,  
    CString& strProgramName  
);
```

Return the name of the last program executed by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszProgramName

A pointer to a string buffer that will contain the name of the last program executed by the client. This parameter cannot be NULL and should be at least 32 characters in size. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This parameter must have a value greater than zero.

Return Value

If the method succeeds, the return value is the length of the command name. If the client ID does not specify a valid client session, the method will return zero. If the client has not executed any programs, this method will return zero.

Remarks

The **GetProgramName** method returns the name of the last program that was executed by the client using the SITE EXEC command. The name that is returned is the alias assigned to the program, not the full path to the executable file. The server application would typically use this method in an event handler when processing the FTP_CLIENT_EXECUTE event to determine which program has been executed on behalf of the client. The **GetProgramExitCode** method will return the program's exit code and the **GetProgramOutput** method can be used to obtain a copy of the output generated by the program.

Permission to use the SITE EXEC is not granted to authenticated users by default, and is limited to only those programs which are explicitly registered with the server. Exercise caution when allowing a client to execute a program on the server because this can expose the server to significant security risks. The programs that are registered for use with the SITE EXEC command should be thoroughly tested before being deployed on the server and should only be console programs that write to standard output.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetProgramExitCode](#), [GetProgramOutput](#), [OnExecute](#), [RegisterProgram](#)

CFtpServer::GetProgramOutput Method

```
DWORD GetProgramOutput(  
    UINT nClientId,  
    LPBYTE lpBuffer,  
    DWORD dwBufferSize  
);
```

Return a copy of the standard output from the last program executed by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpBuffer

A pointer to a buffer that will contain the output from the last program executed by the client. If this parameter is NULL, the method will return the number of bytes of data that was output by the program. Note that this output is not null terminated.

dwBufferSize

The maximum number of bytes that can be copied into the buffer. If the *lpBuffer* parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of bytes copied into the specified buffer. If the client ID does not specify a valid client session, the method will return zero. If the client has not executed any programs, the return value will be zero.

Remarks

The **GetProgramOutput** method is used to obtain a copy of the output generated by the program executed using the SITE EXEC command. To determine the number of bytes of output available to read, call this method with the *lpBuffer* parameter as NULL and the *dwBufferSize* parameter with a value of zero. The return value will be the number of bytes of data that was output by the program. It should be noted that for Unicode builds, the buffer is a byte array, not an array of characters, and will not be null terminated.

This method returns the raw output from the command which may contain escape sequences, control characters and embedded nulls. When the application processes the output returned by this method, it should never coerce the buffer pointer to an LPTSTR value because there is no guarantee that the data will be null-terminated. To obtain the output from the command as a string, use the **GetProgramText** method.

Example

```
LPBYTE lpBuffer = NULL; // A pointer to the output buffer  
DWORD cbBuffer = 0;     // Number of bytes in the output buffer  
  
// Determine the number of bytes in the output buffer  
cbBuffer = pFtpServer->GetProgramOutput(nClientId, NULL, 0);  
  
if (cbBuffer > 0)  
{  
    // Allocate memory for the buffer  
    lpBuffer = new BYTE[cbBuffer + 1];
```

```
// Copy the program output to the buffer
cbBuffer = pFtpServer->GetProgramOutput(nClientId, lpBuffer, cbBuffer + 1);
}

// Free the memory allocated for the buffer when finished
if (lpBuffer != NULL)
{
    delete lpBuffer;
    lpBuffer = NULL;
    cbBuffer = 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[GetProgramExitCode](#), [GetProgramName](#), [GetProgramText](#), [OnExecute](#), [RegisterProgram](#)

CFtpServer::GetProgramText Method

```
INT GetProgramText(  
    UINT nClientId,  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);
```

```
INT GetProgramText(  
    UINT nClientId,  
    CString& strBuffer  
);
```

Return a copy of the standard output from the last program in a string buffer.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszBuffer

A pointer to a buffer that will contain the output from the last program executed by the client as a string. If this parameter is NULL, the method will return the number of bytes of characters that was output by the program, not including a terminating null character. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

The maximum number of bytes that can be copied into the buffer. If the *lpszBuffer* parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of characters copied into the specified string buffer, not including the terminating null character. If the client ID does not specify a valid client session, the method will return zero. If the client has not executed any programs, the return value will be zero.

Remarks

The **GetProgramText** method is used to obtain a copy of the output generated by the program executed using the SITE EXEC command. To determine the number of characters of output available to read, call this method with the *lpszBuffer* parameter as NULL and the *nMaxLength* parameter with a value of zero. The return value will be the number of characters that were output by the program. If the application dynamically allocates the string buffer, make sure that it allocates an extra character for the terminating null character.

This method will only return textual output from the command and any non-printable control characters and the escape character will be replaced with a space. To obtain the unfiltered output from the last command that was executed, use the **GetProgramOutput** method.

Example

```
CString strBuffer;  
  
if (pFtpServer->GetProgramText(nClientId, strBuffer) > 0)  
    pEditCtrl->SetWindowText(strBuffer);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetProgramExitCode](#), [GetProgramName](#), [GetProgramOutput](#), [OnExecute](#), [RegisterProgram](#)

CFtpServer::GetRenamedFile Method

```
INT GetRenamedFile(  
    UINT nClientId,  
    LPTSTR lpszFileName,  
    INT nMaxLength  
);
```

```
INT GetRenamedFile(  
    UINT nClientId,  
    CString& strFileName  
);
```

Return the original name of a file being renamed by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A pointer to a string buffer that will contain the full path of the last file or directory that was renamed. It is recommended that this buffer be at least MAX_PATH characters in size. The value of this parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null characters. This parameter must have a value larger than zero.

Return Value

If the method succeeds, the return value is the length of the file or directory path, not including the terminating null character. If the client ID does not specify a valid client session, the method will return zero. If the client has not renamed a file or directory, this method will return zero.

Remarks

When a client wishes to rename a file or directory, it must send two commands in sequence to the server. The first command is RNFR (rename from) which specifies the original name of the file or directory to be renamed. The second command is RNTD (rename to) and must be sent immediately after the RNFR command and specifies the new name for the file or directory. The **GetRenamedFile** method will return the full path of the local file or directory that was specified by the RNFR command. Typically this method is used by an event handler that processes the **OnCommand** event to determine the original path name.

This method is only guaranteed to return a meaningful value when called within the context of the **OnCommand** event handler. Calling this method outside of the event handler will return the path of the last renamed file, but there is no way to determine at what point the client issued the command to rename a file or directory. To obtain the new file or directory name, the **GetCommandFile** method should be called from within the event handler.

The RNFR and RNTD commands can also be used to move a file or directory to a new location. For example, they could be used to move a file from one directory to another. An application should never make the assumption that the paths of the original and new file will be the same.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetCommandFile](#), [OnCommand](#)

CFtpServer::GetStackSize Method

```
DWORD GetStackSize();
```

Return the initial size of the stack allocated for threads created by the server.

Parameters

None.

Return Value

If the method succeeds, the return value is the amount of memory that will be allocated for the stack in bytes. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetStackSize** method returns the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cstdlib10.h

Import Library: csftsv10.lib

See Also

[SetStackSize](#), [Start](#)

CFtpServer::GetTransferInfo Method

```
BOOL GetTransferInfo(  
    UINT nClientId,  
    LPFTPTRANSFER LpTransferInfo  
);
```

Return information about the current file transfer for the client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpTransferInfo

A pointer to an **FTPTRANSFER** structure that will contain information about the last file transfer. This parameter cannot be NULL, and the *dwSize* member of the structure must be initialized to specify the structure size prior to calling this method.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero. This method should only be called after the client has issued the APPE, RETR, STOR or STOU commands to initiate a file transfer, otherwise the return value will be zero.

Remarks

The **GetTransferInfo** method is used to obtain information about the last file transfer that was performed by the client. This method is typically called within an event handler to determine how many bytes of data were transferred, the type of file and the full path to the file on the local system.

If the default event handler is used, the **OnDownload** and **OnUpload** methods will be invoked with information about the transfer passed as arguments to the handler.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetCommandResult](#), [OnDownload](#), [OnUpload](#), [FTPTRANSFER](#)

CFtpServer::GetUuid Method

```
INT GetUuid(  
    LPTSTR lpszHostUuid,  
    INT nMaxLength  
);  
  
INT GetUuidString(  
    CString& strHostUuid  
);
```

Return the UUID assigned to the server as a printable string.

Parameters

lpszHostUuid

A pointer to a string buffer that will contain the server UUID, terminated with a null character. It is recommended that this buffer be at least 40 characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the buffer is not large enough to store the complete UUID string, the method will return a value of zero.

Remarks

The **GetUuid** method returns the Universally Unique Identifier (UUID) that has been assigned to the server. The UUID may either be generated by the application and assigned as part of the server configuration, or an ephemeral UUID may be automatically generated when the server is started.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SetUuid](#)

CFtpServer::IsActive Method

```
BOOL IsActive();
```

Determine if the server has been started.

Return Value

This method returns a non-zero value if the server has been started. If the server is stopped this method will return zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[CFtpServer](#), [IsListening](#), [Start](#), [Stop](#)

CFtpServer::IsClientAnonymous Method

```
BOOL IsClientAnonymous(  
    UINT nClientId  
);
```

Determine if the specified client has authenticated as an anonymous user.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the client session has been authenticated, this method will return a non-zero value, otherwise it will return zero. If the client ID is valid, and the client session has been authenticated, this method will clear the last error code.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[AuthenticateClient](#), [GetClientCredentials](#), [IsClientAuthenticated](#)

CFtpServer::IsClientAuthenticated Method

```
BOOL IsClientAuthenticated(  
    UINT nClientId  
);
```

Determine if the specified client session has been authenticated.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the client session has been authenticated, this method will return a non-zero value, otherwise it will return zero. If the client ID is valid, this method will clear the last error code.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[AuthenticateClient](#), [GetClientCredentials](#), [IsClientAnonymous](#)

CFtpServer::IsCommandEnabled Method

```
BOOL IsCommandEnabled(  
    LPCTSTR lpszCommand  
);
```

Determine if a specific server command has been enabled or disabled.

Parameters

lpszCommand

A pointer to a NULL terminated string that specifies the name of the command. The command name is not case-sensitive, but the value must otherwise match the exact command name.

Partial matches are not recognized by this method. This parameter cannot be NULL.

Return Value

If the command is enabled, this method will return a non-zero value. If the command is disabled or the command name does not match a supported command, this method will return zero.

Remarks

The **IsCommandEnabled** method is used to determine whether a specific command is enabled. Typically this method is used in an event handler to make sure the command issued by a client is recognized by the server and enabled for use. Commands can be enabled using the **EnableCommand** method and disabled using the **DisableCommand** method.

This method does not account for the permissions granted to a specific client session. Clients are assigned access rights when they are authenticated using the **AuthenticateClient** method, and certain commands can be limited by the permissions granted to the client. For example, even though the STOR command is enabled, a client must have the FTP_ACCESS_WRITE permission to use the command to upload a file to the server. For a list of access rights, see [User Access Constants](#).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AuthenticateClient](#), [DisableCommand](#), [EnableCommand](#), [GetCommandName](#)

CFtpServer::IsInitialized Method

BOOL IsInitialized();

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[CFtpServer](#)

CFtpServer::IsListening Method

```
BOOL IsListening();
```

The **IsListening** method returns whether or not the server is listening for client connections.

Return Value

This method returns a non-zero value if the server has been started and is listening for client connections. If the server is stopped or has been suspended this method will return zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[CFtpServer](#), [Start](#), [Stop](#)

CFtpServer::PreProcessEvent Method

```
virtual LONG PreProcessEvent(  
    HSERVER hServer,  
    UINT nClientId,  
    UINT nEventId,  
    DWORD dwError,  
    BOOL& bHandled  
);
```

A virtual method that is invoked for each event generated by the server.

Parameters

hServer

The server handle. The application should treat this as an opaque value that is only valid as long as the server is active. This value should not be stored by the application and the handle value will change if the server is restarted.

nClientId

An unsigned integer which uniquely identifies the client that has issued a request to the server. This value is guaranteed to be unique to the client session throughout the life of the server and is never reused. The application should never make assumptions about the order in which IDs are allocated to the client sessions.

nEventId

An unsigned integer which specifies which event occurred. For a list of events, see [Server Event Constants](#).

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

bHandled

An integer which specifies if the event has been handled by the application. If this parameter is set to a non-zero value, the default event handler will not be invoked for the event.

Return Value

The method should return a value of zero to indicate that the default event handler should be invoked for the event. If the method returns a non-zero value, this value is passed back to the event dispatcher and the default handler will not be invoked.

Remarks

The **PreProcessEvent** method is invoked for each event that is generated, prior to the default handler for that event. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

CFtpServer::RegisterProgram Method

```
BOOL RegisterProgram(  
    LPCTSTR LpszCommandName,  
    LPCTSTR LpszProgramFile  
);
```

```
BOOL RegisterProgram(  
    LPCTSTR LpszCommandName,  
    LPCTSTR LpszProgramFile,  
    LPCTSTR LpszParameters,  
    LPCTSTR LpszDirectory,  
);
```

Register a program for use with the SITE EXEC command.

Parameters

LpszCommandName

A pointer to a string which specifies the name of the site specific command. This is the name that is passed to the SITE EXEC command and does not need to match the actual name of the executable file on the local system. The maximum length of the command name is 32 characters. This parameter cannot be NULL.

LpszProgramFile

A pointer to a string that specifies the full path to the executable program. This parameter cannot be NULL.

LpszParameters

A pointer to a string that specifies additional parameters for the program. If the program does not require any command line parameters, this value may be NULL or point to an empty string.

LpszDirectory

A pointer to a string that specifies the current working directory for the program. If this parameter is NULL or points to an empty string, the server will use the current working directory of the client that issues the SITE EXEC command.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **RegisterProgram** method registers an executable program for use with the SITE EXEC command. Because this can present a significant security risk to the server, clients are not given permission to use this command by default. A client must be explicitly granted permission to use SITE EXEC by including FTP_ACCESS_EXECUTE as one of the permissions when authenticating the client session with the **AuthenticateClient** method or creating a virtual user using the **AddVirtualUser** method.

To give the server complete control over what programs can be executed using SITE EXEC, the program must be registered with the server and referenced by an alias specified by the *LpszCommandName* parameter. The maximum length of a program name is 31 characters and it must be at least 3 characters in length. The name must only consist of alphanumeric characters and the first character of the program name cannot be numeric. The program name is not case-sensitive, however convention is to use upper-case characters. If a program name is specified that

already has been registered, it will be updated with the new information provided by this method.

The *lpszProgramFile* string specifies file name of the program that will be executed. You should not install any executable programs in the server root directory or its subdirectories. A client should never have the ability to directly access the executable file itself. It is permitted to have multiple command names that reference the same executable file. The only requirement is that the command names be unique. The program name may contain environment variables surrounded by % symbols. For example, %ProgramFiles% would be expanded to the **C:\Program Files** folder.

It is important to note that the program specified by *lpszProgramFile* must be an executable file, not a script or batch file. If the program name does not contain a directory path, then the standard Windows pathing rules will be used when searching for an executable file that matches the given name. It is recommended that you always provide a full path to the executable file.

The *lpszParameters* string is used to define optional command line parameters that will be included with the command. This string can contain placeholders that are replaced by additional parameters specified by the client when it sends the SITE EXEC command. First replacement parameter is %1, the second is %2 and so on.

The executable program that is registered using this method must be a console application that writes to standard output. Programs that write directly to a console, or programs written to use a Windows user interface are not supported and will yield unpredictable results. In most cases, those programs that do not use standard input and output will be forcibly terminated by the server. If the program attempts to read from standard input, it will immediately encounter an end-of-file condition. Programs executed by the SITE EXEC command have no input; it is similar to a program that has its input redirected from the NUL: device. If the program must process a file on the server, the local file name should be passed as a command line parameter.

The output from the program will be redirected back to the client control channel. The output should be textual, with each line of text terminated by a carriage return and linefeed (CRLF). Programs that write binary data to standard output, particular data with embedded nulls, will yield unpredictable results and are not supported. To ensure that the program output conforms to the protocol standard, any non-printable characters will be replaced with a space and each line of output will be prefixed by a single space. The server application can obtain a copy of the output from the last command by calling the **GetProgramOutput** method.

If the server is running on a system with User Account Control (UAC) enabled and does not have elevated privileges, do not register a program that requires elevated privileges or has a manifest that specifies the requestedExecutionLevel as requiring administrative privileges.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetProgramExitCode](#), [GetProgramName](#), [GetProgramOutput](#)

CFtpServer::RenameServerLogFile Method

```
BOOL RenameServerLogFile(  
    LPCTSTR lpszFileName  
);
```

Rename or delete the current log file being updated by the server.

Parameters

lpszFileName

A pointer to a string that specifies the file name the current log file should be renamed to. If this parameter is NULL or an empty string, the current log file will be deleted.

Return Value

If the method succeeds, the return value is non-zero. If logging is not currently enabled for the server, this method will return zero.

Remarks

The **RenameServerLogFile** method is used to rename or delete the current log file. Note that this does not change the current log file name or disable logging by the server. It only changes the file name of the current log file, or removes the log file if the *lpszFileName* parameter is NULL. This can be useful if you want your server to perform log file rotation, archiving the current log file. By renaming the current log file, the server will automatically create a new log file with original file name.

This method must be used to rename or delete the current log file while logging is active because the server holds an open handle on the file. The application should not use the **GetLogFile** method to obtain the log file name and then use the **MoveFileEx** or **DeleteFile** Windows API functions with that file.

To disable logging, use the **SetLogFile** method and specify the logging format as FTP_LOGFILE_NONE.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLogFile](#), [SetLogFile](#)

CFtpServer::Restart Method

```
BOOL Restart();
```

Restart the server, terminating all active client sessions.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Restart** method will restart the specified server, terminating all active client sessions. If the method is unable to restart the server for any reason, the server thread is terminated. The server retains all of the configuration parameters from the previous instance, however the statistical information (such as the number of clients, files transferred, etc.) will be reset.

If an application calls this method from within an event handler, the active client session (the client for which the event handler was invoked) may not get a disconnect notification. It is recommended that this method only be called by the same thread that created the server using the **Start** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[Start](#), [Stop](#)

CFtpServer::Resume Method

BOOL Resume();

Resume accepting client connections.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Resume** method instructs the server to resume accepting new client connections after the **Suspend** method has been called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[Restart](#), [Start](#), [Stop](#), [Suspend](#), [Throttle](#)

CFtpServer::SendResponse Method

```
BOOL SendResponse(  
    UINT nClientId,  
    UINT nResultCode,  
    LPCTSTR lpszMessage  
);
```

Send a result code and message to the client in response to a command.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nResultCode

An unsigned integer value which specifies the result code.

lpszMessage

A pointer to a string which specifies a message to be sent to the client. If this parameter is NULL or points to an empty string, a default message associated with the result code will be used.

Return Value

If the result code and message text was sent to the client, the return value is non-zero. If the client ID does not specify a valid client session, or the result code is invalid, this method will return zero.

Remarks

The **SendResponse** method is used to respond to a command issued by the client. Command responses are normally handled by the server as a normal part of processing a command and this method is only used if the application has implemented custom commands or wishes to modify the standard responses sent by the server. The message may be a maximum of 2048 characters and may include embedded carriage-return and linefeed characters. If no message is specified, then a default message will be sent based on the result code.

Result codes must be three digits (in the range of 100 through 999) and although this method will support the use of non-standard result codes, it is recommended that the client application use the standard codes defined in RFC 959 whenever possible. The use of non-standard result codes may cause problems with FTP clients that expect specific result codes in response to a particular command. For more information, refer to the **GetCommandResult** method.

This method should only be called once in response to a command sent by the client. If a result code has already been sent in response to a command and this method is called, it will fail and return a value of zero. This is necessary because sending multiple result codes in response to a single command may cause unpredictable behavior by the client.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetCommandResult](#)

CFtpServer::SetAddress Method

```
BOOL SetAddress(  
    LPCTSTR lpszAddress  
);
```

Change the IP address that the server will use with passive data connections.

Parameters

lpszAddress

A pointer to a string that specifies the IP address that the server should use for passive mode data transfers. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If either the server handle or the IP address is invalid the method will return a value of zero.

Remarks

The **SetAddress** method changes the IP address that the server will use when a client transfers a file using a passive mode data connection. In passive mode, the server will create a second passive (listening) socket that will accept an incoming connection from the client. The server sends the IP address and port number allocated for that socket to the client, and the client establishes the data connection to the server using that address. The IP address that the server sends to the client is normally the same as the IP address that the client used to establish the control connection, however if the server is located behind a router that performs Network Address Translation (NAT), the IP address reported to the client may not be usable.

This method enables your application to set the external IP address for the server to a specific value, rather than the server attempting to automatically discover its own external address. If you wish to set the external address for the server manually, call the **Start** method without the FTP_SERVER_EXTERNAL option and then call this method to set the external IP address to the desired value.

This method will not change the IP address the server is using to listen for client connections. The only way to change the listening IP address is to stop and restart the server using the new address. This method only changes the IP address that is reported to clients when a passive data connection is used. Incorrect use of this method can prevent the client from establishing a data connection to the server. The address must be in the same address family as the local address that the server was started with. For example, if the server was started using an IPv4 address, the IP address passed to this method cannot be an IPv6 address.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetAddress](#), [GetName](#)

CFtpServer::SetCertificate Method

```
BOOL SetCertificate(  
    DWORD dwProtocol,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName,  
    LPCTSTR lpszPassword  
);
```

```
BOOL SetCertificate(  
    LPCTSTR lpszCertName,  
    LPCTSTR lpszPassword  
);
```

Set the name of the certificate to be used with secure connections.

Parameters

dwProtocol

An unsigned integer that specifies the security protocols to be used when establishing a secure connection with the client. This parameter may be one or more of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default selection of security protocols will be used when establishing a connection. The TLS 1.2, TLS 1.1 and TLS 1.0 protocols will be negotiated with the server, in that order of preference. This option will always request the latest version of the preferred security protocols and is the recommended value.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Note that SSL 2.0 has been deprecated and will never be used unless the server does not support version 3.0.
SECURITY_PROTOCOL_TLS	The TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.

lpszCertStore

A pointer to a string which specifies the name of certificate store. This may be the name of the certificate store in registry, or it may specify the name of a file that contains the certificate and

its private key.

lpzCertName

A pointer to a string which specifies the common name for the certificate that will be used. Typically this will be the fully qualified domain name for the server.

lpzPassword

An optional pointer to a string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only required if the *lpzCertStore* parameter specifies a certificate file in PKCS12 format that is password protected.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **SetCertificate** method will create the security credentials required for the server to accept secure connections and enable security options for the server. This method will not validate the certificate information provided by the application. If the certificate does not exist, or does not have a private key associated with it, the client will be unable to establish a secure connection.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetOptions](#), [SetOptions](#)

CFtpServer::SetClientAccess Method

```
BOOL SetClientAccess(  
    UINT nClientId,  
    DWORD dwUserAccess  
);
```

Change the access rights associated with the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies one or more user access rights. For a list of user access rights that can be granted to the client, see [User Access Constants](#).

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero. This method can only be used with authenticated clients. If the client session has not been authenticated, the return value will be zero.

Remarks

The **SetClientAccess** method can change the access rights for an authenticated client session. This method can only be used after the **AuthenticateClient** method has been used to grant the initial set of access rights to the client. The **EnableClientAccess** method can be used to grant or revoke a specific permission for the client session.

The *dwUserAccess* parameter has a value of `FTP_ACCESS_DEFAULT`, then default permissions will be granted to the client session. A normal client cannot be changed to a restricted or anonymous client using this method. If the `FTP_ACCESS_RESTRICTED` or `FTP_ACCESS_ANONYMOUS` access flags are specified, this method will fail.

This method cannot be used to change the access rights for a restricted or anonymous user. Those rights are granted when the client session is authenticated and will persist until the client disconnects from the server. This restriction is designed to prevent the inadvertent granting of rights to an untrusted client that could compromise the security of the server.

Example

```
DWORD dwUserAccess = 0;  
  
// Allow the client to execute programs using SITE EXEC  
if (pFtpServer->GetClientAccess(nClientId, dwUserAccess))  
{  
    if (! (dwUserAccess & FTP_ACCESS_ANONYMOUS))  
        pFtpServer->SetClientAccess(nClientId, dwUserAccess |  
FTP_ACCESS_EXECUTE);  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftsv10.lib`

See Also

[AuthenticateClient](#), [EnableClientAccess](#), [GetClientAccess](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CFtpServer::SetClientFileType Method

```
BOOL SetClientFileType(  
    UINT nClientId,  
    UINT nFileType  
);
```

Change the current file type used for transfers by the specified client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nFileType

Specifies the type of file that will be uploaded or downloaded. This parameter determines whether subsequent file transfers require any data conversion and may be one of the following values.

Value	Description
FILE_TYPE_ASCII (1)	The file is a text file using the ASCII character set. For those clients which use a different end-of-line character sequence, the text file has been converted to the local format which uses the carriage return (CR) and linefeed (LF) characters.
FILE_TYPE_IMAGE (3)	The file is a binary file and no data conversion of any type has been performed on the file. This is the default file type for most data files and executable programs. If the client specified this file type when appending to a text file, the file will contain the end-of-line sequences used by its native operating system.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **SetClientFileType** method will change the default file type that is used for subsequent transfers by the client. If the file type is set to FILE_TYPE_ASCII then the server will automatically convert any end-of-line character sequences to match the format used by the local system. For example, if the client is connecting from a UNIX based system, the server will convert a single linefeed character to a carriage return (CR) and linefeed (LF) sequence. If the file type is set to FILE_TYPE_IMAGE, then no conversion is performed.

This method can be used to override the file type specified by filtering the TYPE command issued by the client. For example, it could be used to force file transfers to use a specific type based on the file extension, regardless of the type specified by the client. To determine the current file type set by the client, use the **GetClientFileType** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

See Also

[GetClientFileType](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CFtpServer::SetClientIdentity Method

```
BOOL SetClientIdentity(  
    UINT nClientId,  
    LPCTSTR lpszIdentity  
);
```

Change the identity of the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszIdentity

A pointer to a string that identifies the client. If this parameter is NULL or specifies an empty string, the current identity for the client is cleared. The maximum length of the identity string is 64 characters, including the terminating null character.

Return Value

If the method succeeds, the return value is non-zero. If the client ID is invalid, the method will return a value of zero.

Remarks

The **SetClientIdentity** method associates a string value with the client that can be used to identify the session. The identity string does not have any standard format and is used for informational purposes only. Typically it is used to identify the client application that was used to establish the connection. Changing the client identity has no effect on the operation of the server. To obtain the identity string currently associated with the client, use the **GetClientIdentity** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientIdentity](#), [GetIdentity](#), [SetIdentity](#)

CFtpServer::SetClientIdleTime Method

```
UINT SetClientIdleTime(  
    UINT nClientId,  
    UINT nTimeout  
);
```

Change the idle timeout period for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nTimeout

An unsigned integer value that specifies the number of seconds that the client may remain idle. If this value is zero, the default idle timeout period for the server will be used.

Return Value

If the method succeeds, the return value is the previous client idle timeout period in seconds. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **SetClientIdleTime** method will change the number of seconds that the client may remain idle before being automatically disconnected by the server. The minimum timeout period for a client is 60 seconds, the maximum is 7200 seconds (2 hours). The idle time of a client session is based on the last time a command was issued to the server or when a data transfer completed.

If the value INFINITE is specified as the timeout period, the client activity timer will be refreshed, extending the idle timeout period for the session. This is typically done inside an **OnTimeout** event handler to prevent the client from being disconnected due to inactivity.

To obtain the current idle timeout period for a client, along with the amount of time the client has been idle, use the **GetClientIdleTime** method.

This timeout period only affects authenticated clients. Unauthenticated clients use a different internal timer that limits the amount of time they can remain connected to the server and that value cannot be changed for individual client sessions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[GetClientIdleTime](#)

CFtpServer::SetCommandFile Method

```
BOOL SetCommandFile(  
    UINT nClientId,  
    LPCTSTR lpszFileName  
);
```

Change the name of the local file or directory that is the target of the current command.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A pointer to a string that specifies the new file name. This parameter may be NULL to specify that the original file or directory name should be used.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **SetCommandFile** method is used by the application to change the target file or directory name for the current command from within the **OnCommand** event handler. This can be used to effectively redirect the client to use a different file than the one that was actually requested. For example, if the client issues the RETR command to download a file from the server, this method can be used to redirect the command to use a different file name. To obtain the full path to the file or directory that is the target of the current command, use the **GetCommandFile** method.

The *lpszFileName* parameter specifies the path to the new file or directory name. If the path is absolute, then it will be used as-is. If the path is relative, it will be relative to the current working directory for the client session. The full path to this file is not limited to the server root directory or its subdirectory, it can specify a file anywhere on the local system. If this parameter is a NULL pointer, or points to an empty string, then the server will revert to using the actual file or directory name specified by the command. This enables the application to effectively undo a previous call to this method to change the target file name.

Typically this method would be used to redirect a client to a file or directory that it may not normally have access to. Exercise caution when using this method to provide access to data that is stored outside of the server root directory. Incorrect use of this method could expose the server to security risks or cause unpredictable behavior by client applications.

This method should only be called within the context of the **OnCommand** event handler, and only for those commands that perform an action on a file or directory. If the current command does not target a file or directory, this method will return zero and the last error code will be set to ST_ERROR_INVALID_COMMAND. To obtain the name of the current command issued by the client, use the **GetCommandName** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetCommandFile](#) [GetCommandLine](#), [GetCommandName](#)

CFtpServer::SetDirectory Method

```
BOOL SetDirectory(  
    LPCTSTR lpszDirectory  
);
```

Specify the local directory that will be used as the server root directory.

Parameters

lpszDirectory

A pointer to a string that specifies the root directory for the server. If this parameter is NULL or a zero-length string, the server will use the current working directory as the root directory.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return zero.

Remarks

The **SetDirectory** method specifies the path to the local directory that should be used as the root document directory for the server.

It is recommended that you always specify an absolute path for the server root directory. If the path includes environment variables surrounded by percent (%) symbols, they will be automatically expanded.

If you have configured the server to permit clients to upload files, you must ensure that your application has permission to create files in the directory that you specify. A recommended location for the server root directory would be a subdirectory of the %ALLUSERSPROFILE% directory. Using the environment variable ensures that your server will work correctly on different versions of Windows. If the root directory does not exist at the time that the server is started, it will be created.

You cannot change the server root directory after the server has started. To change the root directory, you must stop the server using the **Stop** method and then start another instance of the server with a configuration that specifies the new directory.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetDirectory](#)

CFtpServer::SetIdentity Method

```
BOOL SetIdentity(  
    LPCTSTR lpszIdentity  
);
```

Change the identity of the specified server.

Parameters

lpszIdentity

A pointer to a string that identifies the server. If this parameter is NULL or specifies an empty string, the current identity for the server is reset to a default value. The maximum length of the identity string is 64 characters, including the terminating null character.

Return Value

If the method succeeds, the return value is non-zero, otherwise it will return a value of zero.

Remarks

The **SetClientIdentity** method changes a string value used by the server to identify itself to clients. The identity string does not have any standard format and is used for informational purposes only. Typically it consists of the application name and a version number. Changing the server identity has no effect on the operation of the server. To obtain the identity string currently associated with the server, use the **GetIdentity** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientIdentity](#), [SetClientIdentity](#), [GetIdentity](#)

CFtpServer::SetLastError Method

```
VOID SetLastError(  
    DWORD dwError  
);
```

Set the last error code for the specified server session.

Parameters

dwError

An unsigned integer that specifies an error code.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each server session. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SERVER or FTP_ERROR.

If the *dwError* parameter is specified with a value of zero, this effectively clears the error code for the last method that failed. Those methods which clear the last error code when they succeed are noted on their reference page.

Applications can retrieve the value saved by this method by calling the **GetLastError** method to determine the specific reason for failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[GetCommandResult](#), [GetLastError](#), [SendResponse](#)

CFtpServer::SetLogFile Method

```
BOOL SetLogFile(  
    UINT nLogFormat,  
    UINT nLogLevel,  
    LPCTSTR lpszFileName  
);
```

Change the current log format, level of detail and file name.

Parameters

nLogFormat

An integer value that specifies the format used when creating or updating the server log file. The following formats are supported:

Constant	Description
FTP_LOGFILE_NONE (0)	This value specifies that the server should not create or update a log file.
FTP_LOGFILE_COMMON (1)	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about file transfers.
FTP_LOGFILE_EXTENDED (2)	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session.

nLogLevel

An integer value that specifies the level of detail that should be generated in the log file. The minimum value is 1 and the maximum value is 10. If this parameter is zero, it is the same as specifying a log file format of FTP_LOGFILE_NONE and will disable logging by the server.

lpszFileName

A pointer to a string that specifies the name of the log file that should be created or appended to. If the server was configured with logging enabled and this parameter is NULL or an empty string, the current log file name will not be changed. If the log file does not exist, it will be created. If it does exist, the contents of the log file will be appended to.

Return Value

If the method succeeds, the return value is non-zero. If one of the parameters are invalid, the method will return zero.

Remarks

The **SetLogFile** method can be used to change the current log file name, the format of the log file or the level of detail recorded in the log file. In some situations it may be desirable to delete the current log file contents when changing the format or ensure that a new log file is created. To do this, combine the *nLogFormat* parameter with the constant FTP_LOGFILE_DELETE.

The higher the value of the *nLogLevel* parameter, the greater the level of detail that is recorded by the server. A log level of 1 instructs the server to only record file transfers, while a level of 10 instructs the server to record all commands processed by the server. Because a higher level of

logging detail can negatively impact the performance of the server, it is recommended that you do not exceed a level of 5 for most applications. A log level of 10 should only be used for debugging purposes.

Example

```
UINT nLogFormat = FTP_LOGFILE_NONE;
UINT nLogLevel = 0;
UINT nNewLevel = 5;
BOOL bChanged = FALSE;

// Change the level of detail for the current log file if logging
// has been enabled and the current level is a lower value

if (pFtpServer->GetLogFile(&nLogFormat, &nLogLevel, NULL, 0))
{
    if (nLogFormat != FTP_LOGFILE_NONE && nLogLevel < nNewLevel)
        bChanged = pFtpServer->SetLogFile(nLogFormat, nNewLevel, NULL);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLogFile](#), [RenameServerLogFile](#)

CFtpServer::SetName Method

```
BOOL SetName(  
    UINT nClientId,  
    LPCTSTR lpszHostName  
);
```

Change the host name assigned to the specified server or client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszHostName

A pointer to a string that specifies the new host name assigned to the server or client session. If this value is NULL or points to an empty string, the current host name will be changed to use the default host name.

Return Value

If the method succeeds, the return value is non-zero. If the client ID is invalid, or the buffer is not large enough to store the complete hostname, the method will return a value of zero.

Remarks

This method will change the host name assigned to the specified client session. If the *nClientId* parameter has a value of zero, the method will change default host name that was assigned to the server as part of the server configuration. If the *nClientId* parameter specifies a valid client session and the *lpszHostName* parameter is NULL, the host name associated with the client session will be changed to the current host name assigned to the server.

When a client connects to the server, it can specify the host name that it used to establish the connection by sending the HOST command. This is typically used with virtual hosting, where one server may accept client connections for multiple domains. The **GetName** method will return the host name specified by the client, and **SetName** can be used by the application to either explicitly assign a different host name to the client session, or override the host name provided by the client.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetAddress](#), [GetName](#)

CFtpServer::SetOptions Method

```
BOOL SetOptions(  
    DWORD dwOptions  
);
```

Sets the default options for this instance of the server.

Parameters

dwOptions

An unsigned integer which specifies one or more options. For a list of available options, see [Server Option Constants](#).

Return Value

If the method is successful, it will return a non-zero value, otherwise it will return a value of zero.

Remarks

The **SetOptions** method changes the default options for the current instance of the server. This method cannot be used to change the options for an active instance of the server. If the server is active, it must be stopped before calling this method. To get the current options, use the **GetOptions** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[GetOptions](#)

CFtpServer::SetPriority Method

```
INT SetPriority(  
    INT nPriority  
);
```

Change the priority assigned to the specified server.

Parameters

nPriority

An integer value which specifies the new priority for the server. It may be one of the following values:

Constant	Description
FTP_PRIORITY_BACKGROUND (0)	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
FTP_PRIORITY_LOW (1)	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
FTP_PRIORITY_NORMAL (2)	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
FTP_PRIORITY_HIGH (3)	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
FTP_PRIORITY_CRITICAL (4)	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Return Value

If the method succeeds, the return value is the previous priority assigned to the server. If the method fails, the return value is FTP_PRIORITY_INVALID. To get extended error information, call the **GetLastError** method.

Remarks

The **SetPriority** method can be used to change the current priority assigned to the specified

server. Client connections that are accepted after this method is called will inherit the new priority as their default priority. Previously existing client connections will not be affected by this method.

Higher priority values increase the thread priority and processor utilization for each client session. You should only change the server priority if you understand the impact it will have on the system and have thoroughly tested your application. Configuring the server to run with a higher priority can have a negative effect on the performance of other programs running on the system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cstools10.h`.

Import Library: `csftsv10.lib`

See Also

[GetPriority](#), [Start](#)

CFtpServer::SetStackSize Method

```
BOOL SetStackSize(  
    DWORD dwStackSize  
);
```

Change the initial size of the stack allocated for threads created by the server.

Parameters

dwStackSize

The amount of memory that will be committed to the stack for each thread created by the server. If this value is zero, a default stack size will be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetStackSize** method changes the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes. Increasing or decreasing the stack size will only affect new threads that are created by the server, it will not affect those threads that have already been created to manage active client sessions. It is recommended that most applications use the default stack size.

You should not change this value unless you understand the impact that it will have on your system and have thoroughly tested your application. Increasing the initial commit size of the stack will remove pages from the total system commit limit, and every page of memory that is reserved for stack cannot be used for any other purpose.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cstdlib10.h

Import Library: csftsv10.lib

See Also

[GetStackSize](#), [Start](#)

CFtpServer::SetUuid Method

```
BOOL SetUuid(  
    LPCTSTR lpszHostUuid  
);
```

Assign a UUID to the current instance of the server.

Parameters

lpszHostUuid

A pointer to a string that specifies the server UUID, terminated with a null character. This value can be used when storing information about the server, and should be generated using a utility such as **uuidgen** which is included with Visual Studio. This parameter may be NULL or point to an empty string, in which case a temporary UUID will be randomly generated for the server.

Return Value

If the method succeeds, the return value is non-zero, otherwise the method return a value of zero.

Remarks

The **SetUuid** method assigns a Universally Unique Identifier (UUID) to the server. The UUID may either be generated by the application and assigned as part of the server configuration, or an ephemeral UUID may be automatically generated when the server is started. This method cannot be used to change the UUID after the server has been started. To determine the UUID assigned to an active server instance, use the **GetUuid** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetUuid](#)

CFtpServer::Start Method

```
BOOL Start(  
    LPCTSTR lpszLocalHost,  
    UINT nLocalPort,  
    DWORD dwOptions  
);
```

```
BOOL Start(  
    LPCTSTR lpszLocalHost,  
    UINT nLocalPort  
);
```

```
BOOL Start(  
    UINT nLocalPort  
);
```

The **Start** method begins listening for client connections on the specified local address and port number. The server is started in its own thread and manages the client sessions independently of the calling thread. All interaction with the server and its client sessions takes place inside the class event handlers.

Parameters

lpszLocalHost

A pointer to a string which specifies the local hostname or IP address address that the server should be bound to. If this parameter is omitted or specifies a NULL pointer an appropriate address will automatically be used. If a specific address is used, the server will only accept client connections on the network interface that is bound to that address.

nLocalPort

The port number the server should use to listen for client connections. If a value of zero is specified, the server will use the standard port number 21 to listen for connections, or port 990 if the server is configured to use implicit SSL. The port number used by the application must be unique and multiple instances of a server cannot use the same port number. It is recommended that a port number greater than 5000 be used for private, application-specific implementations.

dwOptions

An unsigned integer value that specifies one or more options to be used when creating an instance of the server. For a list of the available options, see [Server Option Constants](#). If this parameter is omitted, the default options for the server instance will be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

In most cases, the *lpszLocalHost* parameter should be omitted or a NULL pointer. On a multihomed system, this will enable the server to accept connections on any appropriately configured network adapter. Specifying a hostname or IP address will limit client connections to that particular address. Note that the hostname or address must be one that is assigned to the local system, otherwise an error will occur.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the method will fail.

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

The handle returned by this method references the listening socket that was created when the server was started. The service is managed in another thread, and all interaction with the server and active client connections are performed inside the event handlers. To disconnect all active connections, close the listening socket and terminate the server thread, call the **Stop** method.

The host UUID that is defined as part of the server configuration should be generated using the **uuidgen** utility that is included with the Windows SDK. You should not use the UUID that is provided in the example code, it is for demonstration purposes only. If no host UUID is specified in the server configuration, an ephemeral UUID will be generated automatically when the server is started.

Example

```
CFtpServer *pFtpServer = new CFtpServer();

// Initialize the server configuration
pFtpServer->SetName(_T("server.company.com"));
pFtpServer->SetUuid(_T("10000000-1000-1000-1000-100000000000"));
pFtpServer->SetDirectory(_T("%ProgramData%\\MyProgram\\Server"));
pFtpServer->SetLogFile(FTP_LOGFILE_EXTENDED, 5,
    _T("%ProgramData%\\MyProgram\\Server.log"));
pFtpServer->SetOptions(FTP_SERVER_LOCALUSER | FTP_SERVER_UNIXMODE);

// Start the server
pFtpServer->Start(FTP_PORT_DEFAULT);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnumClients](#), [Restart](#), [Stop](#)

CFtpServer::Stop Method

BOOL Stop();

Stop the server, terminating all active client sessions and releasing the resources that were allocated for the server.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return a value of zero.

Remarks

The **Stop** method instructs the server to stop accepting client connections, disconnects all active client connections and terminates the thread that is managing the server session. The handle is no longer valid after the server has been stopped and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new server is started. An application should always consider the server handle to be opaque and never depend on it being a specific value.

If an application calls this method from within an event handler, the active client session (the client for which the event handler was invoked) may not get a disconnect notification. It is recommended that this method only be called by the same thread that created the server using the **Start** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[Restart](#), [Start](#)

CFtpServer::Suspend Method

```
BOOL Suspend();
```

Suspend the server and reject new client connections.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Suspend** method instructs the server to suspend accepting new client connections. Any incoming client connections will be rejected with an error message indicating that the server is currently unavailable. To resume accepting client connections, call the **Resume** method. Suspending the server will have no effect on clients that have already established a connection with the server.

It is recommended that you only suspend a server if absolutely necessary, and only for brief periods of time. If you want to limit the number of active client connections or control the connection rate for clients, use the **Throttle** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

See Also

[Restart](#), [Resume](#), [Start](#), [Stop](#), [Throttle](#)

CFtpServer::Throttle Method

```
BOOL Throttle(  
    UINT nMaxClients,  
    UINT nMaxClientsPerAddress,  
    UINT nMaxGuests,  
    DWORD dwConnectionRate  
);
```

The **Throttle** method limits the number of active client connections, connections per address and connection rate.

Parameters

nMaxClients

A value which specifies the maximum number of clients that may connect to the server. A value of zero specifies that there is no fixed limit to the number of client connections. A value of -1 specifies that the maximum number of clients should not be changed.

nMaxClientsPerAddress

A value which specifies the maximum number of clients that may connect to the server from the same IP address. A value of zero specifies that there is no fixed limit to the number of client connections per address. By default, there is a limit of four client connections per address. A value of -1 specifies that the maximum number of clients should not be changed.

nMaxGuests

An integer value which specifies the maximum number of guest users that may login to the server. A value of zero disables guest logins and requires that all clients authenticate with a valid username and password. A value of -1 specifies that the maximum number of guest users should not be changed.

dwConnectionRate

A value which specifies a restriction on the rate of client connections, limiting the number of connections that will be accepted within that period of time. A value of zero specifies that there is no restriction on the rate of client connections. The higher this value, the fewer the number of connections that will be accepted within a specific period of time. By default, there is no limit on the client connection rate. A value of -1 specifies that the connection rate should not be changed.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Throttle** method is used to limit the number of connections and the connection rate to minimize the potential impact of a large number of client connections over a short period of time. This can be used to protect the server from a client application that is malfunctioning or a deliberate denial-of-service attack in which the attacker attempts to flood the server with connection attempts.

If the maximum number of client connections or maximum number of connections per address is exceeded, the server will reject subsequent connection attempts until the number of active client sessions drops below the specified threshold. Note that adjusting these values lower than the current connection limits will not affect clients that have already connected to the server. For

example, if the **Start** method is called with the maximum number of clients set to 100, and then **Throttle** is called lowering that value to 75, no existing client connections will be affected by the change. However, the server will not accept any new connections until the number of active clients drops below 75.

Increasing the connection rate value will force the server to slow down the rate at which it will accept incoming client connection requests. For example, setting this parameter to a value of 1000 would limit the server to accepting one client connection every second, while a value of 250 would allow the server to accept four client connections per second. Note that significantly increasing the amount of time the server must wait to accept client connections can exceed the connection backlog queue, resulting in client connections being rejected.

It is recommended that you always specify conservative connection limits for your server application based on expected usage. Allowing an unlimited number of client connections can potentially expose the system to denial-of-service attacks and should never be done for servers that are accessible over the Internet.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[Restart](#), [Resume](#), [Start](#), [Suspend](#)

CFtpServer Event Handlers

Method	Description
OnAuthenticate	The client requested authentication
OnCommand	The client issued a command to the server
OnConnect	The client established a connection to the server
OnDisconnect	The client has disconnected from the server
OnDownload	The client has downloaded a file from the server
OnError	The server encountered an error while handling a client request
OnExecute	The client has executed an external program on the server
OnLogin	The client has successfully authenticated the session
OnLogout	The client has logged out or reinitialized the session
OnResult	The command issued by the client has been processed by the server
OnTimeout	The client has exceeded the maximum allowed idle time
OnUpload	The client has uploaded a file to the server

CFtpServer::OnAuthenticate Method

```
virtual void OnAuthenticate(  
    UINT nClientId,  
    LPCTSTR lpszHostName,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword  
);
```

A virtual method that is invoked after the client has requested authentication.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszHostName

A pointer to a string that specifies the host name that the client used to establish the connection.

lpszUserName

A pointer to a string that specifies the user name provided by the client.

lpszPassword

A pointer to a string that specifies the cleartext user password provided by the client.

Return Value

None.

Remarks

The **OnAuthenticate** event handler is invoked when the client has requested authentication by sending the USER and PASS command to the server. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

The event handler can call the **AuthenticateClient** method to authenticate the client session. To reject an authentication attempt because of an invalid user name or password, the handler should call the **SendResponse** method and specify a result code of 430.

If the client session is not authenticated, the server will perform its default authentication process. If the FTP_SERVER_ANONYMOUS configuration option has been specified, and the client has logged in as an anonymous user, the session will be authenticated as a restricted user. If the FTP_SERVER_LOCALUSER configuration option has been specified, the user name and password is checked against the local user database. If the credentials are valid, the session will be authenticated as a regular user. If neither the FTP_SERVER_ANONYMOUS or FTP_SERVER_LOCALUSER options were specified, the default action is to reject all authentication attempts.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SendResponse](#), [OnLogin](#), [OnLogout](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CFtpServer::OnCommand Method

```
virtual void OnCommand(  
    UINT nClientId,  
    LPCTSTR lpszCommand  
);
```

A virtual method that is invoked after the client has sent a command to the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCommand

A pointer to a string that specifies the command issued by the client. The command name will always be capitalized. For a complete list of commands supported by the server, see [Server Commands](#).

Return Value

None.

Remarks

The **OnCommand** event handler is invoked after the client has sent a command to the server, but before the command has been processed. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

This event handler is invoked for all commands issued by the client, including invalid or disabled commands. If the event handler processes the command, it must call the **SendResponse** method to send a success or error response back to the client. If this is not done, the server will perform the default processing for the command.

Although this event handler will provide the command name, the full command line can be obtained by using the **GetCommandLine** method. Individual command parameters can be obtained by using the **GetCommandParam** and **GetCommandParamCount** methods.

It is not necessary to use this event handler to disable a command. The **EnableCommand** method can be used to enable or disable specific commands, and the **IsCommandEnabled** method can be used to determine if a command is enabled.

If this event handler is used to implement a custom command, it is recommended that you use the **IsClientAuthenticated** method to determine whether or not the client session has been authenticated. Unless there is a specific need for the custom command to be used before a client has logged in, the application should not take any action and send a 530 result code back to the client indicating authentication is required.

Example

```
VOID CMyFtpServer::OnCommand(UINT nClientId, LPCTSTR lpszCommand)  
{  
    // Implement a custom command named TIME that will return the local time  
    if (lstrcmp(lpszCommand, _T("TIME")) == 0)  
    {  
        // The command should not have any parameters  
        if (GetCommandParamCount(nClientId) > 0)  
        {  
            SendResponse(nClientId, FTP_REPLY_BADARG);  
        }  
    }  
}
```

```
        return;
    }

    if (IsClientAuthenticated(nClientId))
    {
        CString strTime = CTime::GetCurrentTime().Format(_T("%Y-%m-%d
%H:%M:%S"));
        SendResponse(nClientId, FTP_REPLY_CMDOK, strTime);
    }
    else
    {
        // The client has not logged in, return an error
        SendResponse(nClientId, FTP_REPLY_NOLOGIN);
    }
}
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisconnectClient](#), [EnableCommand](#), [IsCommandEnabled](#), [OnDisconnect](#), [SendResponse](#)

CFtpServer::OnConnect Method

```
virtual void OnConnect(  
    UINT nClientId,  
    LPCTSTR lpszAddress  
);
```

A virtual method that is invoked after the client has connected to the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszAddress

A pointer to a string that specifies the IP address of the client. This address may either be in IPv4 or IPv6 format, depending on how the server was configured and the address the client used to establish the connection.

Return Value

None.

Remarks

The **OnConnect** event handler is invoked after the client has connected to the server. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

This event only occurs after the server has checked the active client limits and the TLS handshake has been performed, if security has been enabled. If the server has been suspended, or the limit on the maximum number of client sessions has been exceeded, the server will terminate the client session prior to this event handler being invoked.

If this event handler is not implemented, the server will perform the default action of accepting the connection and sending a standard greeting to the client. If you want your application to send a custom greeting to the client when it connects, call the **SendResponse** method, specifying a result code of 220 and a message of your choice.

To reject a connection, call the **SendResponse** method to send an error response to the client. Typically the result code value would be 421 to indicate that the server will not accept the connection. Next, call the **DisconnectClient** method to terminate the client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisconnectClient](#), [OnDisconnect](#), [SendResponse](#)

CFtpServer::OnDisconnect Method

```
virtual void OnDisconnect(  
    UINT nClientId  
);
```

A virtual method that is invoked immediately before the client is disconnected from the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

None.

Remarks

The **OnDisconnect** event handler is invoked immediately before the client is disconnected from the server. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftsv10.lib

See Also

[OnConnect](#), [OnLogout](#)

CFtpServer::OnDownload Method

```
virtual void OnDownload(  
    UINT nClientId,  
    LPCTSTR lpszFileName,  
    DWORD dwTimeElapsed,  
    ULARGE_INTEGER uiBytesCopied  
);
```

A virtual method that is invoked after the client has successfully downloaded a file from the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A pointer to a string that specifies the local path name of the file that was downloaded from the server. The path will always include the disk volume or share name, and the path delimiter will always be the backslash character.

dwTimeElapsed

An unsigned integer value that specifies the number of milliseconds that it took to complete the file transfer.

uiBytesCopied

An unsigned 64-bit integer value that specifies the number of bytes of data that was downloaded by the client.

Return Value

None.

Remarks

The **OnDownload** event handler is invoked after the client has successfully downloaded a file from the server using the RETR command. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

The ULARGE_INTEGER structure is actually a union that is used to represent a 64-bit value. If the compiler has built-in support for 64-bit integers, use the **QuadPart** member to access the 64-bit integer value. Otherwise, use the **LowPart** and **HighPart** members to access the value.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[OnCommand](#), [OnUpload](#)

CFtpServer::OnError Method

```
virtual void OnConnect(  
    UINT nClientId,  
    UINT nEventId,  
    DWORD dwError  
);
```

A virtual method that is invoked when the server encounters an error while handling a client request.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nEventId

An unsigned integer which identifies the client event that was being processed when the error occurred. For a list of event identifiers, see [Server Event Constants](#).

dwError

An unsigned integer value that specifies the error code.

Return Value

None.

Remarks

The **OnError** event handler is invoked whenever an error occurs while an event is being processed by the server. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

It is important to note that this event is not raised for every error that occurs. The event only occurs when another event is being processed and an unhandled error occurs that must be reported back to the server application. The following are some common situations in which this event handler may be invoked:

- A network error occurs when the client connection is being accepted by the server. This could be the result of an aborted connection or some other lower-level failure reported by the networking subsystem on the server.
- The server is configured to use implicit SSL but cannot obtain the security credentials required to create the security context for the session. Usually this indicates that the server certificate cannot be found, or the certificate does not have a private key associated with it. It could also indicate a general problem with the cryptographic subsystem where the client and server could not successfully negotiate a cipher suite.
- A network error occurs when attempting to process a command issued by the client. This usually indicates that the connection to the client has been aborted, either because the client is not acknowledging the data that has been exchanged with the server, or the client has terminated abnormally. This event will not occur if the client terminates the connection normally.

In most situations where this event handler is invoked, the error is not recoverable and the only action that can be taken is to terminate the client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[OnConnect](#), [OnCommand](#)

CFtpServer::OnExecute Method

```
virtual void OnExecute(  
    UINT nClientId,  
    LPCTSTR lpszProgram,  
    DWORD dwExitCode  
);
```

A virtual method that is invoked after the client has successfully executed an external program on the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszProgram

A pointer to a string that specifies the name of the program that was executed. This is the registered program name, and not a full path to the executable and its command arguments.

dwExitCode

An unsigned integer that specifies the exit code that was returned by the program.

Return Value

None.

Remarks

The **OnExecute** event handler is invoked after the client has successfully executed an external program using the SITE EXEC command. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

This event will only be generated if the client has the FTP_ACCESS_EXECUTE permission. Clients are not granted this permission by default, and must be explicitly permitted to execute external programs. If the client does have this permission, it can only execute specific programs that have been registered by the server application using the **RegisterProgram** method.

The **GetProgramOutput** method can be used to obtain the unfiltered output from the external command, while the **GetProgramText** method will return filtered output from the program that contains only printable text characters.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetProgramOutput](#), [GetProgramText](#), [RegisterProgram](#)

CFtpServer::OnLogin Method

```
virtual void OnLogin(  
    UINT nClientId,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszDirectory,  
    LPCTSTR dwUserAccess  
);
```

A virtual method that is invoked after the client has successfully authenticated the session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszUserName

A pointer to a string that specifies the user name.

lpszDirectory

A pointer to a string that specifies the full path to the home directory for the client. The path will always include the disk volume or share name, and the path delimiter will always be the backslash character.

dwUserAccess

An unsigned integer which specifies one or more access rights for the client session. For a list of user access rights that can be granted to the client, see [User Access Constants](#).

Return Value

None.

Remarks

The **OnLogin** event handler is invoked after the client has successfully authenticated itself using the USER and PASS commands. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

To convert the home directory path to a virtual path name that is relative to the server root directory, use the **GetClientVirtualPath** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AddVirtualUser](#), [AuthenticateClient](#), [OnAuthenticate](#), [OnLogout](#)

CFtpServer::OnLogout Method

```
virtual void OnLogout(  
    UINT nClientId,  
    LPCTSTR lpszUserName  
);
```

A virtual method that is invoked after the client has logged out or reinitialized the session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszUserName

A pointer to a string that specifies the user name.

Return Value

None.

Remarks

The **OnLogout** event handler is invoked after the client has successfully logged out using the QUIT command or reinitialized the session using the REIN command. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

The application should not depend on this event handler always being invoked when a client is disconnected from the server. This event only occurs when the client sends the QUIT or REIN commands and will not be invoked if the client connection is aborted or disconnected for some other reason, such as exceeding the idle timeout period. If the application needs to update data structures or perform some cleanup when a client disconnects, that should be done in the **OnDisconnect** event handler.

The application should not call the **DisconnectClient** method in the handler for this event because the client is either in the process of disconnecting or expects that it can submit new credentials to the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[OnConnect](#), [OnDisconnect](#), [OnLogin](#)

CFtpServer::OnResult Method

```
virtual void OnResult(  
    UINT nClientId,  
    LPCTSTR lpszCommand,  
    UINT nResultCode  
);
```

A virtual method that is invoked after the server has processed a command issued by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCommand

A pointer to a string that specifies the command that was issued by the client. This value contains only the command and not the additional parameters that may have been specified.

nResultCode

An integer value that specifies the result code that was sent to the client.

Return Value

None.

Remarks

The **OnResult** event handler is invoked after the server has processed a command issued by the client. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

To obtain the complete command line, use the **GetCommandLine** method, or use the **GetCommandParam** method to get the value of specific command parameters. If the application requires the text message that was sent to the client along with the result code, use the **GetCommandResult** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetCommandLine](#), [GetCommandParam](#), [GetCommandResult](#), [OnCommand](#)

CFtpServer::OnTimeout Method

```
virtual void OnTimeout(  
    UINT nClientId,  
    UINT nIdleTime,  
    UINT nElapsed  
);
```

A virtual method that is invoked after the client has exceeded the maximum allowed idle time.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nIdleTime

An unsigned integer value that specifies the current idle timeout period for the client in seconds.

nElapsed

An unsigned integer value that specifies the number of seconds that the client has been idle.

Return Value

None.

Remarks

The **OnTimeout** event handler is invoked after the client has exceeded the maximum allowed idle time. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

This event handler will be invoked prior to the client being disconnected from the server. This event will never occur during a file transfer or directory listing. The **SetClientIdleTime** method can be used to change or refresh the idle timeout period for the session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[GetClientIdleTime](#), [OnCommand](#), [OnResult](#), [SetClientIdleTime](#)

CFtpServer::OnUpload Method

```
virtual void OnUpload(  
    UINT nClientId,  
    LPCTSTR lpszFileName,  
    DWORD dwTimeElapsed,  
    ULARGE_INTEGER uiBytesCopied  
);
```

A virtual method that is invoked after the client has successfully uploaded a file to the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A pointer to a string that specifies the local path name of the file that was uploaded to the server. The path will always include the disk volume or share name, and the path delimiter will always be the backslash character.

dwTimeElapsed

An unsigned integer value that specifies the number of milliseconds that it took to complete the file transfer.

uiBytesCopied

An unsigned 64-bit integer value that specifies the number of bytes of data that was uploaded by the client.

Return Value

None.

Remarks

The **OnUpload** event handler is invoked after the client has successfully uploaded a file to the server using the APPE, STOR or STOU command. To implement an event handler, the application should create a class derived from the **CFtpServer** class, and then override this method.

The ULARGE_INTEGER structure is actually a union that is used to represent a 64-bit value. If the compiler has built-in support for 64-bit integers, use the **QuadPart** member to access the 64-bit integer value. Otherwise, use the **LowPart** and **HighPart** members to access the value.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[OnCommand](#), [OnDownload](#)

CFtpServer Data Structures

- FTPCLIENTCREDENTIALS
- FTPSERVERTRANSFER

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

FTPCLIENTCREDENTIALS Structure

The **FTPCLIENTCREDENTIALS** structure defines the credentials used to authenticate a specific user.

```
typedef struct _FTPCLIENTCREDENTIALS
{
    DWORD dwSize;
    DWORD dwFlags;
    TCHAR szHostName[FTP_MAXHOSTNAME];
    TCHAR szUserName[FTP_MAXUSERNAME];
    TCHAR szPassword[FTP_MAXPASSWORD];
} FTPCLIENTCREDENTIALS, *LPFTPCLIENTCREDENTIALS;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwFlags

An unsigned integer value reserved for future use. This member will always be initialized to a value of zero.

szHostName

A pointer to a string that specifies the server host name.

szUserName

A pointer to a string that specifies the user name.

szPassword

A pointer to a string that specifies the user password.

Remarks

When an instance of this structure is passed to the **GetClientCredentials** method, this member must be initialized to the size of the structure and all other members must be initialized with a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientCredentials](#)

FTPSEVERTRANSFER Structure

The **FTPSEVERTRANSFER** structure provides information about the last file transfer performed by a client.

```
typedef struct _FTPSEVERTRANSFER
{
    DWORD          dwSize;
    DWORD          dwReserved;
    DWORD          dwFileAccess;
    DWORD          dwTimeElapsed;
    ULARGE_INTEGER uiBytesCopied;
    TCHAR          szFileName[MAX_PATH];
} FTPSEVERTRANSFER, *LPFTPSEVERTRANSFER;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwReserved

An unsigned integer value that is reserved for future use. This value will always be zero.

dwFileAccess

An unsigned integer value that specifies the how the local file was accessed. It can be one of the following values:

Constant	Description
FTP_FILE_READ (0)	The file was opened for reading. This mode indicates that the client issued the RETR command to download the contents of a file from the server to the client system. The <i>szFileName</i> member specifies the name of the local file on the server that was downloaded by the client.
FTP_FILE_WRITE (1)	The file was opened for writing. This mode indicates that the client issued the STOR or STOU command to upload the contents of a file from the client system to server. The <i>szFileName</i> member specifies the name of the local file on the server that was created by the client. If a file already existed with the name name, it was replaced.
FTP_FILE_APPEND (2)	The file was opened for writing. This mode indicates the client issued the APPE command to upload the contents of a file from the client system and append the data to a file on the server. If the file did not exist, then it was created. The <i>szFileName</i> member specifies the name of the local file that was appended to or created by the client.

dwTimeElapsed

The amount of time that it took for the file transfer to complete in milliseconds. This value is limited to the resolution of the system timer, which is typically in the range of 10 to 16 milliseconds. This value may be zero if the transfer occurred over a local network or on the same host using a loopback address.

uiBytesCopied

A 64-bit integer value that specifies the total number of bytes copied during the file transfer. This value is represented by a `ULARGE_INTEGER` union which provides support for those programming languages that do not have intrinsic support for 64-bit integers. For more information, refer to the Windows SDK documentation. The application should not make the assumption that this is the actual size of the file. If the client specified a restart offset using the `REST` command, this value would only represent the number of bytes transferred from that byte offset, not the total file size.

szFileName

A pointer to a string value that will contain the full path to the local file that was transferred. The ***dwFileAccess*** member determines whether the file name represents a file that was downloaded by the client, or uploaded from the client and stored on the server.

Remarks

When an instance of this structure is passed to the **GetTransferInfo** method, the ***dwSize*** member must be initialized to the size of the structure, otherwise the method will fail with an error indicating that the parameter is invalid. All other members should be initialized to a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetTransferInfo](#)

Hypertext Transfer Protocol Class Library

Transfer files between the local system and a web server, execute scripts and perform remote file management functions.

Reference

- [Class Methods](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

Class Name	CHttpClient
File Name	CSHTPV10.DLL
Version	10.0.1468.2518
LibID	FAA72D93-4063-474E-97DE-25BA304E1098
Import Library	CSHTPV10.LIB
Dependencies	None
Standards	RFC 1945, RFC 2616

Overview

The Hypertext Transfer Protocol (HTTP) is a lightweight, stateless application protocol that is used to access resources on web servers, as well as send data to those servers for processing. The library provides direct, low-level access to the server and the commands that are used to retrieve resources (i.e.: documents, images, etc.). The library also provides a simple interface for downloading resources to the local host, similar to how the FTP library can be used to download files.

In a typical session, the library is used to establish a connection, send a request (to download a resource, post data for processing, etc.), read the data returned by the server and then disconnect. It is the responsibility of the client to process the data returned by the server, depending on the type of resource that was requested.

This library supports secure connections using the standard SSL and TLS protocols.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Hypertext Transfer Protocol Class Methods

Class	Description
CHttpClient	Constructor which initializes the current instance of the class
~CHttpClient	Destructor which releases resources allocated by the class
Method	Description
AddField	Append the form field and its value to the current form
AddFile	Append the contents of the file to the current form
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified client handle to another thread
Authenticate	Specify authentication information for restricted resources
Cancel	Cancel the current blocking operation
ClearForm	Remove all defined fields from the current form
CloseFile	Close the file opened on the server
Command	Send a command to the server
Connect	Connect to the specified server
ConnectUrl	Establish a client connection using the specified URL
CreateFile	Create or replace a file on the server
CreateForm	Create a new form to submit to the server
CreateSecurityCredentials	Create a new security credentials structure
DeleteFile	Remove a file from the server
DeleteField	Delete the form field and its value from the current form
DeleteHeaders	Delete all of the response or request headers for the current session
DeleteSecurityCredentials	Delete a previously created security credentials structure
DestroyForm	Destroy the current form and free the memory allocated for it
DetachHandle	Detach the handle for the current instance of this class
DisableEvents	Disable asynchronous event notification
DisableTrace	Disable logging of socket function calls to the trace log
Disconnect	Disconnect from the current server
DownloadFile	Download a file from the server to the local system
EnableCompression	Enable or disable support for data compression
EnableEvents	Enable asynchronous event notification
EnableTrace	Enable logging of socket function calls to a file
FreezeEvents	Suspend asynchronous event processing
GetBearerToken	Return the current OAuth 2.0 bearer token for the client session

GetCookie	Return information about the specified cookie
GetData	Copy the specified resource to a local buffer
GetEncodingType	Determines which content encoding option is enabled
GetErrorString	Return a description for the specified error code
GetFile	Copy a file from the server to the local system
GetFileSize	Return the size of a file on the server
GetFileTime	Return the date and time a file on the server was last modified
GetFirstCookie	Return the first cookie set by the server
GetFirstHeader	Return the name and value of the first request or response header field
GetFormProperties	Return the properties of the specified form
GetHandle	Return the client handle used by this instance of the class
GetHeader	Return the value of the specified response header field
GetLastError	Return the last error code
GetNextCookie	Return the next cookie set by the server
GetNextHeader	Return the name and value of the next request or response header field
GetOption	Return the enabled/disabled state of a specified option
GetPriority	Return the current priority for file transfers
GetResultCode	Return the result code from the previous command
GetResultString	Return the result string from the previous command
GetSecurityInformation	Return security information about the current client connection
GetStatus	Return the current client status
GetText	Download the contents of a text file or resource to a string buffer
GetTimeout	Return the number of seconds until an operation times out
GetTransferStatus	Return data transfer statistics
HttpEventProc	Callback method that processes events generated by the client
IsBlocking	Determine if the client is blocked, waiting for information
IsConnected	Determine if the client is connected to the server
IsInitialized	Determine if the class has been successfully initialized
IsReadable	Determine if data can be read from the server
IsWritable	Determine if data can be written to the server
OpenFile	Open a file on the server for reading
PatchData	Submits JSON or XML patch data to the server and returns the response
PostData	Post data from a local buffer to the server and returns the response
PostFile	Submit the contents of a local file to the server

PostJson	Post JSON formatted data to the server and returns the response
PostXml	Post XML formatted data to the server and returns the response
ProxyConnect	Establish a connection with the specified proxy server
PutData	Create a file on the server using the contents of a local buffer
PutFile	Copy a file from the local system to the server
PutText	Create a text file on the server from the contents of a string buffer
Read	Read data from the server
RegisterEvent	Register an event callback function
SetBearerToken	Set the value of the OAuth 2.0 bearer token for the client session
SetCookie	Set the value of the specified cookie
SetEncodingType	Specifies the type of encoding to be applied to data submitted to the server
SetFormProperties	Modify the properties of the current form
SetHeader	Set the value of a request header field
SetLastError	Set the last error code
SetOption	Enable or disable the specified option
SetPriority	Set the priority for file transfers
SetTimeout	Set the number of seconds until an operation times out
ShowError	Display a message box with a description of the specified error
SubmitForm	Submit the current form to the server for processing
UploadFile	Upload a file from the local system to the server
ValidateUrl	Check the contents of a string to ensure it represents a valid URL
VerifyFile	Compare the size of a local file against a file stored on the server
Write	Write data to the server

CHttpClient::CHttpClient

`CHttpClient();`

The **CHttpClient** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **HttpInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[~CHttpClient](#), [IsInitialized](#)

CHttpClient::~~CHttpClient

`~CHttpClient();`

The **CHttpClient** destructor releases resources allocated by the current instance of the **CHttpClient** object. It also uninitialized the library if there are no other concurrent uses of the class.

Remarks

When a **CHttpClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshttpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CHttpClient](#)

CHttpClient::AddField Method

```
INT AddField(  
    LPCTSTR lpszFieldName,  
    LPVOID lpFieldData,  
    DWORD cbFieldData  
);
```

```
INT AddField(  
    LPCTSTR lpszFieldName,  
    LPCTSTR lpszFieldData  
);
```

The **AddField** method adds a new field to the current form.

Parameters

lpszFieldName

A pointer to a string which specifies the name of the field to add to the form. If this parameter is NULL or points to an empty string, then a default field name will be assigned.

lpFieldData

A pointer to the form field data. Typically this will either be by a pointer to an array of bytes or a string which specifies the value for the form field. If no data is to be associated with the form field, then this argument may be NULL.

cbFieldData

An unsigned integer value which specifies the number of bytes of data in the form field. If this value is 0xFFFFFFFF (-1) then it is assumed that the *lpFieldData* parameter is a pointer to a null-terminated string. If *lpFieldData* is NULL, this value must be zero.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **AddField** method is used to add a field and its associated value to a form created using the **CreateForm** method. If the field name has already been added to the form, the previous value is deleted and replaced by the new value.

Example

```
CString strResult;  
INT nResult = 0;  
  
pClient->CreateForm(_T("/login.php"), HTTP_METHOD_POST, HTTP_FORM_ENCODED);  
pClient->AddField(_T("UserName"), lpszUserName);  
pClient->AddField(_T("Password"), lpszPassword);  
  
nResult = pClient->SubmitForm(strResult);  
pClient->DestroyForm();
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AddFile](#), [ClearForm](#), [CreateForm](#), [DeleteField](#), [SubmitForm](#)

CHttpClient::AddFile Method

```
INT AddFile(  
    LPCTSTR lpszFieldName,  
    LPCTSTR lpszFileName  
);
```

The **AddFile** method adds the contents of a file to the specified form.

Parameters

lpszFieldName

A pointer to a string which specifies the name of the field to add to the form. If this parameter is NULL or points to an empty string, then a default field name will be assigned.

lpszFileName

A pointer to a string which specifies the name of the file. The contents of the file will be added to the form data that is submitted to the server.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AddField](#), [ClearForm](#), [CreateForm](#), [DeleteField](#), [PostFile](#), [SubmitForm](#)

CHttpClient::AttachHandle Method

```
VOID AttachHandle(  
    HCLIENT hClient  
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle already has been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the *hClient* parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

See Also

[AttachThread](#), [DetachHandle](#), [GetHandle](#)

CHttpClient::AttachThread Method

```
DWORD AttachThread(  
    DWORD dwThreadId  
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

See Also

[AttachHandle](#), [Cancel](#), [Connect](#), [DetachHandle](#), [Disconnect](#), [GetHandle](#)

CHttpClient::Authenticate Method

```
INT Authenticate(  
    UINT nAuthType,  
    LPCTSTR LpszUserName,  
    LPCTSTR LpszPassword  
);
```

```
INT Authenticate(  
    LPCTSTR LpszUserName,  
    LPCTSTR LpszPassword  
);
```

Parameters

nAuthType

An unsigned integer value which specifies the method to be used when authenticating the client. If this argument is not specified, then basic authentication will be used. The following values are recognized.

Constant	Description
HTTP_AUTH_NONE	No client authentication should be performed. The <i>lpszUserName</i> and <i>lpszPassword</i> parameters are ignored and current authentication settings are cleared.
HTTP_AUTH_BASIC	The Basic authentication scheme should be used. This option is supported by all servers that support at least version 1.0 of the protocol. The user credentials are not encrypted and Basic authentication should not be used over standard (non-secure) connections. Most web services which use Basic authentication require the connection to be secure.
HTTP_AUTH_BEARER	The Bearer authentication scheme should be used. This authentication method does not require a user name and the <i>lpszPassword</i> parameter must specify the OAuth 2.0 bearer token issued by the service provider. If the access token has expired, the request will fail with an authorization error. This function will not automatically refresh an expired token.

lpszUserName

A pointer to a string that specifies the username used to authenticate the client session. This parameter may be NULL or an empty string if a user name is not required for the specified authentication type.

lpszPassword

A pointer to a string that specifies the password used to authenticate the client session.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method will set the **Authorization** request header for the client session using the credentials provided by the caller. This method will always override any custom

Authorization

header value that may have been previously set using the **SetRequestHeader** method.

If both the *lpszUserName* and *lpszPassword* parameters are NULL pointers or specify zero length strings, the current authentication type will always be set to HTTP_AUTH_NONE regardless of the value of the *nAuthType* parameter. This effectively clears the current user credentials for the client session.

If the web service requires OAuth 2.0 authentication, it is recommended you use the **SetBearerToken** method to specify the access token.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [GetBearerToken](#), [SetBearerToken](#), [SetHeader](#)

CHttpClient::Cancel Method

```
INT Cancel();
```

The **Cancel** method cancels any outstanding blocking operation in the client, causing the blocking method to fail. The application may then retry the operation or terminate the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

See Also

[IsBlocking](#)

CHttpClient::ClearForm Method

```
VOID ClearForm();
```

The **ClearForm** method clears the current form, removing all fields.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

See Also

[AddField](#), [AddFile](#), [CreateForm](#), [DeleteField](#), [DestroyForm](#), [SubmitForm](#)

CHttpClient::CloseFile Method

```
INT CloseFile();
```

The **CloseFile** method flushes the internal client buffers and closes the previously opened file on the server.

Parameters

None.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the file is opened for writing, all buffered data is written to the server before the file is closed. This may cause the client to block until all of the data can be written. The client application should not perform any other action until the method returns.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

See Also

[Command](#), [OpenFile](#), [Read](#), [Write](#)

CHttpClient::Command Method

```
INT Command(  
    LPCTSTR lpszCommand,  
    LPCTSTR lpszResource,  
    LPBYTE lpParameter,  
    DWORD cbParameter  
);
```

```
INT Command(  
    LPCTSTR lpszCommand,  
    LPCTSTR lpszResource,  
    LPCTSTR lpszParameter  
);
```

The **Command** method sends a command to the server, and returns the result code back to the caller. This method is typically used for extended commands not directly supported by the API.

Parameters

lpszCommand

A pointer to a string which specifies the command to be executed by the server. The following table lists the standard commands recognized by most HTTP servers. Other commands may also be used, such as those extensions used by WebDAV to edit and manage files on a server.

Command	Description
GET	Return the contents of the specified resource. This command is recognized by all servers.
HEAD	Return only header information for the specified resource. This command is recognized by servers that support at least version 1.0 of the protocol.
POST	Post data to the specified resource. This command is recognized by servers that support at least version 1.0 of the protocol.
PUT	Create or replace the specified resource on the server. This command is recognized by servers that support at least version 1.0 of the protocol. Not all servers support this command.
DELETE	Delete the specified resource from the server. This command is recognized by servers that support at least version 1.1 of the protocol. Not all servers support this command.

lpszResource

A pointer to a string which specifies the resource to be used with the command. This can be the name of a file, an executable script or any other valid resource name recognized by the server. Resource names must be absolute and include the complete path to the resource.

lpParameter

A pointer to a byte array which contains data that is to be passed to the command as one or more parameters. Typically this is used to pass additional information to a script that executes on the server. The data is encoded according to the encoding type specified for the client session. If the resource does not require any parameters, this value should be NULL.

cbParameter

Specifies the number of bytes stored in the parameter buffer. If the resource does not require

any parameters, this value should be zero.

Return Value

If the method succeeds, the return value is the result code returned by the server. If the method fails, the return value is `HTTP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

Not all servers support all of the listed commands, and some commands may require specific changes to the server configuration. In particular, the PUT and DELETE commands typically require that configuration changes be made by the site administrator. All servers will support the use of the GET command, and all servers that support at least version 1.0 of the protocol will support the POST command.

If the *lpSzResource* parameter specifies a path that contains reserved or restricted characters, such as a space, it will automatically be URL encoded by the library.

It is permissible to include a query string in the resource name specified by the *lpSzResource* parameter. Query strings begin with a question mark, and then are followed by one or more name/value pairs separated by an equal sign. For example, the following resource includes a query string:

```
/cgi-bin/test.cgi?field1=value1&field2=value2
```

In this case, the query string is "?field1=value1&field2=value2". If the query string contains reserved or restricted characters, such as spaces, then it will be automatically URL encoded prior to being sent to the server. If additional resource data is specified in the *lpParameter* argument along with a query string in the resource name, the action taken by the library depends on the command being sent. If the command is a POST or PUT command, then query string is included with command request to the server and the parameter data is sent separately. For example, if the POST command was used, the script running on the server would see that both query data and form data has been provided to it. However, if any other command is specified, the parameter data is simply appended to the query string.

The *lpParameter* argument is used to pass additional information to the server when a resource is requested. This is most commonly used to provide information to scripts, similar to how arguments are used when executing a program from the command line. Unless the POST command is being executed, the data in the buffer will automatically be encoded using the current encoding mechanism specified for the client. By default, the data is URL encoded, which means that any spaces and non-printable characters are converted to printable characters before submitted to the server. The type of encoding that is performed can be set by calling the **SetEncodingType** method. Although the default encoding is appropriate for most applications, those that submit XML formatted data may need to change the encoding type.

Only one request may be in progress at one time for each client session. Use the **CloseFile** method to terminate the request after all of the data has been read from the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshttpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

CHttpClient::Connect Method

```
BOOL Connect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    DWORD dwVersion,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **Connect** method is used to establish a connection with the server.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to. This may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 80. For secure connections, the default port number is 443.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested. If the server does not support the keep-alive option, the client will automatically reconnect when each resource is requested. Although it will not provide any performance benefits, this allows the option to be used with all servers.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the

	current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed using the system Control Panel.
HTTP_OPTION_ERRORDATA	This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a

	connection using IPv6 regardless if this option has been specified.
HTTP_OPTION_FREETHREAD	This option specifies that this instance of the class may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the class instance is synchronized across multiple threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

dwVersion

The requested protocol version used when sending requests to the server. The high word should specify the major version, and the low word should specify the minor version number. The HTTPVERSION macro can be used to create version value.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur. If this argument is NULL, then the client session will be blocking and no network events will be sent to the client.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the *hEventWnd* argument is NULL, this argument should be specified as WM_NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

If you specify an event notification window, then the client session will be asynchronous. When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
HTTP_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
HTTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
HTTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.

HTTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
HTTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
HTTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
HTTP_EVENT_PROGRESS	The client is in the process of sending or receiving data from the server. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.
HTTP_EVENT_REDIRECT	This event is generated when a the server indicates that the requested resource has been moved to a new location. The new resource location may be on the same server, or it may be located on another server.

The use of Windows event notification messages has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

To prevent this method from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **Connect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

To cancel asynchronous notification and return the client to a blocking mode, use the **DisableEvents** method.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

The *dwOptions* argument can be used to specify the threading model that is used by the class when a connection is established. By default, the client session is initially attached to the thread that created it. From that point on, until the connection is terminated, only the owner may invoke methods in that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the HTTP_OPTION_FREETHREAD option enables any thread to call methods in any instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the class and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a method in the class, it must ensure that no other thread will call another method at the same time using the same instance.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Disconnect](#), [IsConnected](#), [ProxyConnect](#)

CHttpClient::ConnectUrl Method

```
BOOL ConnectUrl(  
    LPCTSTR lpszURL,  
    UINT nTimeout,  
    DWORD dwOptions  
);
```

The **ConnectUrl** method establishes a connection with the specified server using a URL.

Parameters

lpszURL

A pointer to a string which specifies the URL for the server. The URL must follow the conventions for the Hypertext Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested. If the server does not support the keep-alive option, the client will automatically reconnect when each resource is requested. Although it will not provide any performance benefits, this allows the option to be used with all servers.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server,

	then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed using the system Control Panel.
HTTP_OPTION_ERRORDATA	This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
HTTP_OPTION_FREETHREAD	This option specifies that this instance of the class may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the class instance is synchronized across multiple threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller files being uploaded or downloaded using fast network connections.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ConnectUrl** method uses an HTTP URL to establish a connection with a server. The URL must be in the following format:

```
[http|https]://[username : password] @[remotehost] [:remoteport] / [path / ...] [filename]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "http" scheme is used and the HTTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, the "https" scheme must be specified. The **ValidateUrl** method can be used to verify that a URL is valid prior to calling this method.

The **ConnectUrl** method is designed to provide a simpler, more convenient interface to establishing a connection with a server. However, complex connections such as those using a specific proxy server or a secure connection which uses a client certificate will require the program to use the lower-level connection methods. If you only need to upload or download a file using a URL, then refer to the **UploadFile** and **DownloadFile** methods.

The *dwOptions* argument can be used to specify the threading model that is used by the class when a connection is established. By default, the client session is initially attached to the thread that created it. From that point on, until the connection is terminated, only the owner may invoke methods in that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the HTTP_OPTION_FREETHREAD option enables any thread to call methods in any instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the class and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a method in the class, it must ensure that no other thread will call another method at the same time using the same instance.

Example

```
CHttpClient httpClient;  
  
if (!httpClient.ConnectUrl(_T("http://sockettools.com/")))  
{  
    httpClient.ShowError();  
    return;  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [Disconnect](#), [DownloadFile](#), [UploadFile](#), [ValidateUrl](#)

CHttpClient::CreateFile Method

```
INT CreateFile(  
    LPCTSTR lpszRemoteFile,  
    DWORD dwFileLength  
);
```

The **CreateFile** method creates the specified file on the server.

Parameters

lpszRemoteFile

Points to a string that specifies the name of the file being created on the server. The client must have the appropriate access rights to create the file or an error will be returned.

dwFileLength

Specifies the length of the file that will be created on the server. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **CreateFile** method uses the PUT command to create the file. The server must support this command and the user must have the appropriate permission to create the specified file. If this method is successful, the client should then use the **Write** method to send the contents of the file to the server. Once all of the data has been written, the **CloseFile** method should be called to close the file and complete the operation. Note that this method is typically only accepted by servers that support version 1.1 of the protocol or later.

When using **Write** to send the contents of the file to the server, it is recommended that the data be written in logical blocks that are no larger than 8,192 bytes in size. Attempting to write very large amounts of data in a single call can either cause the thread to block or, in the case of an asynchronous connection, return an error if the internal buffers cannot accommodate all of the data. To send the entire contents of a file in a single method call, use the **PutData** method instead of calling **CreateFile**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CloseFile](#), [OpenFile](#), [PutData](#), [PutFile](#), [Write](#)

CHttpClient::CreateForm Method

```
BOOL CreateForm(  
    LPCTSTR lpszAction,  
    UINT nFormMethod,  
    UINT nFormType  
);
```

The **CreateForm** method creates a new form for use with the other form-related functions.

Parameters

lpszAction

A pointer to a string which specifies the name of the resource that the form data will be submitted to. Typically this is the name of a script that is executed on the server.

nFormMethod

An unsigned integer value which specifies how the form data will be submitted to the server. This parameter may be one of the following values:

Constant	Description
HTTP_METHOD_DEFAULT	The form data should be submitted using the default method, using the GET command.
HTTP_METHOD_GET	The form data should be submitted using the GET command. This method should be used when the amount of form data is relatively small. If the total amount of form data exceeds 2048 bytes, it is recommended that the POST method be used instead.
HTTP_METHOD_POST	The form data should be submitted using the POST command. This is the preferred method of submitting larger amounts of form data. If the total amount of form data exceeds 2048 bytes, it is recommended that the POST method be used.

nFormType

An unsigned integer value which specifies the type of form and how the data will be encoded when it is submitted to the server. This parameter may be one of the following values:

Constant	Description
HTTP_FORM_DEFAULT	The form data should be submitted using the default encoding method.
HTTP_FORM_ENCODED	The form data should be submitted as URL encoded values. This is typically used when the GET method is used to submit the data to the server.
HTTP_FORM_MULTIPART	The form data should be submitted as multipart form data. This is typically used when the POST method is used to submit a file to the server. Note that the script must understand how to process multipart form data if this form type is specified.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **CreateForm** method is used to create a new form that will be populated with values and then submitted to the server for processing. When the form is no longer needed, it should be destroyed using the **DestroyForm** function.

Example

```
CString strResult;
INT nResult = 0;

pClient->CreateForm(_T("/login.php"), HTTP_METHOD_POST, HTTP_FORM_ENCODED);
pClient->AddField(_T("UserName"), lpszUserName);
pClient->AddField(_T("Password"), lpszPassword);

nResult = pClient->SubmitForm(strResult);
pClient->DestroyForm();
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AddField](#), [AddFile](#), [ClearForm](#), [DeleteField](#), [DestroyForm](#), [SubmitForm](#)

CHttpClient::CreateSecurityCredentials Method

```
BOOL CreateSecurityCredentials(  
    DWORD dwProtocol,  
    DWORD dwOptions,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName  
);  
  
BOOL CreateSecurityCredentials(  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName  
);  
  
BOOL CreateSecurityCredentials(  
    LPCTSTR lpszCertName  
);
```

The **CreateSecurityCredentials** method establishes the security credentials for the client session.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols.

	This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpzUserName

A pointer to a string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A pointer to a string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A pointer to a string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Example

```
pClient->CreateSecurityCredentials(  
    SECURITY_PROTOCOL_DEFAULT,  
    0,  
    NULL,  
    NULL,  
    lpszCertStore,  
    lpszCertName);
```

```
bConnected = pClient->Connect(lpszHostName,  
                              HTTP_PORT_SECURE,  
                              HTTP_TIMEOUT,  
                              HTTP_OPTION_SECURE);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [DeleteSecurityCredentials](#), [GetSecurityInformation](#), [SECURITYCREDENTIALS](#)

CHttpClient::DeleteField Method

```
INT DeleteField(  
    LPCTSTR lpszFieldName  
);
```

The **DeleteField** method deletes the specified field from the current form.

Parameters

lpszFieldName

Points to a string that specifies the name of the field to delete.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AddField](#), [AddFile](#), [ClearForm](#), [CreateForm](#), [SubmitForm](#)

CHttpClient::DeleteFile Method

```
INT DeleteFile(  
    LPCTSTR lpszFileName  
);
```

The **DeleteFile** method deletes the specified file from the server.

Parameters

lpszFileName

Points to a string that specifies the name of the remote file to delete. The file pathing and name conventions must be that of the server.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method uses the DELETE command to delete the specified file from the server. The server must be configured to support this command, and client must have the appropriate permission to delete the file, or an error will be returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFile](#), [PutFile](#)

CHttpClient::DeleteHeaders Method

```
BOOL DeleteHeaders(  
    UINT nHeaderType  
);
```

Delete all of the response or request headers for the current session.

Parameters

nHeaderType

Specifies the type of header to delete. It may be one of the following values:

Constant	Description
HTTP_HEADERS_REQUEST	Delete all of the request headers that have been set for the next request sent to the server. This will clear all of the header values that were set using the SetRequestHeader method.
HTTP_HEADERS_RESPONSE	Delete all of the headers that were set in response to the previous request. A call to the GetResponseHeader method to obtain a specific header value will fail after this method returns.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **DeleteHeaders** method will release the memory allocated for the request or response headers for the current session. This method is typically used to clear all of the current request headers so that the application may create a new set of headers when a persistent connection is being used. The memory allocated for the request and response headers is normally released when the session handle is closed by calling the **Disconnect** method.

Whenever a request for a resource is sent to the server, the response headers from the previous request are automatically cleared. It is not necessary for an application to call the **DeleteHeaders** method to delete the response headers prior to each request.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

See Also

[GetFirstHeader](#), [GetNextHeader](#), [GetHeader](#), [SetHeader](#)

CHttpClient::DeleteSecurityCredentials Method

```
VOID DeleteSecurityCredentials();
```

The **DeleteSecurityCredentials** method releases the security credentials for the current session.

Parameters

None.

Return Value

None.

Remarks

This method can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated. The security credentials are released when the class destructor is called, so it is normally not required that the application explicitly call this method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateSecurityCredentials](#)

CHttpClient::DestroyForm Method

```
VOID DestroyForm();
```

The **DestroyForm** method destroys the current form, releasing the memory allocated for it.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

See Also

[AddField](#), [AddFile](#), [ClearForm](#), [CreateForm](#), [DeleteField](#), [SubmitForm](#)

CHttpClient::DetachHandle Method

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshttpv10.lib`

See Also

[AttachHandle](#), [GetHandle](#)

CHttpClient::DisableEvents Method

```
INT DisableEvents();
```

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the client has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

See Also

[EnableEvents](#), [RegisterEvent](#)

CHttpClient::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

See Also

[EnableTrace](#)

CHttpClient::Disconnect Method

VOID Disconnect();

The **Disconnect** method terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

See Also

[Connect](#)

CHttpClient::DownloadFile Method

```
BOOL DownloadFile(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszFileURL,  
    UINT nTimeout  
    DWORD dwOptions  
    LPHTTPTRANSFERSTATUS lpStatus  
    HTTPEVENTPROC lpEventProc  
    DWORD_PTR dwParam  
);
```

The **DownloadFile** method downloads the specified file from the server to the local system.

Parameters

lpszLocalFile

A pointer to a string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszFileURL

A pointer to a string that specifies the complete URL of the file to be downloaded. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed using the system Control Panel.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server

	certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.

lpStatus

A pointer to an HTTPTRANSFERSTATUS structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **EventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **DownloadFile** method provides a convenient way for an application to download a file in a single method call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session and then download the file to the local system. The URL must be complete, and specify either a standard or secure HTTP scheme:

```
[http|https]://[username : password] @] remotehost [:remoteport] / [path / ...] [filename]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "http" scheme is used and the HTTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, the "https" scheme must be specified.

The *lpStatus* parameter can be used by the application to determine the final status of the transfer, including the total number of bytes copied, the amount of time elapsed and other information related to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The *lpEventProc* parameter specifies a pointer to a function which will be periodically called during the file transfer process. This can be used to check the status of the transfer by calling **GetTransferStatus** and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The *dwParam* parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the *this* pointer as the value, and then cast it back to an object pointer inside the callback function. If no event handler is required, then a NULL pointer can be specified as the value for *lpEventProc* and the *dwParam* parameter will be ignored.

The **DownloadFile** method is designed to provide a simpler interface for downloading a file. However, complex connections such as those using a specific proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **Connect** and then use **GetFile** to download the file.

Example

```
CHttpClient httpClient;  
CString strLocalFile = _T("c:\\temp\\database.mdb");  
CString strFileURL = _T("http://www.example.com/updates/database.mdb");  
  
// Download the file using the specified URL  
if (!httpClient.DownloadFile(strLocalFile, strFileURL))  
{  
    httpClient.ShowError();  
    return;  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFile](#), [GetTransferStatus](#), [HttpEventProc](#), [UploadFile](#), [HTTPTRANSFERSTATUS](#)

CHttpClient::EnableCompression Method

```
INT EnableCompression(  
    BOOL bEnable  
);
```

The **EnableCompression** method enables or disables support for data compression.

Parameters

bEnable

An optional boolean value which specifies if data compression should be enabled or disabled. A non-zero value enables compression, while a value of zero will disable compression. If this parameter is omitted, compression will be enabled.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpEnableCompression** method is used to indicate to the server whether or not it is acceptable to compress the data that is returned to the client. If compression is enabled, the client will advertise that it will accept compressed data by setting the **Accept-Encoding** request header. The server will decide whether a resource being requested can be compressed. If the data is compressed, the library will automatically expand the data before returning it to the caller.

Enabling compression does not guarantee that the data returned by the server will actually be compressed, it only informs the server that the client is willing to accept compressed data. Whether or not a particular resource is compressed depends on the server configuration, and the server may decide to only compress certain types of resources, such as text files. Disabling compression informs the server that the client is not willing to accept compressed data; this is the default.

If the **SetHeader** method is used to explicitly set the **Accept-Encoding** header to request compressed data and compression is not enabled, the library will not attempt to automatically expand the data returned by the server. In this case, the raw compressed data will be returned to the caller and the application is responsible for processing it. This behavior is by design to maintain backwards compatibility with previous versions of the library that did not have internal support for compression.

To determine if the server compressed the data returned to the client, use the **GetHeader** method to get the value of the **Content-Encoding** header. If the header is defined, the value specifies the compression method used, otherwise the data was not compressed.

Enabling compression is only meaningful when downloading files from a server that supports file compression. It has no effect on file uploads.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

See Also

CHttpClient::EnableEvents Method

```
INT EnableEvents(  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Constant	Description
HTTP_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
HTTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
HTTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking

	operation. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
HTTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
HTTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
HTTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

It is not required that the client be placed in asynchronous mode in order to receive command and progress event notifications. To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

See Also

[DisableEvents](#), [RegisterEvent](#)

CHttpClient::EnableTrace Method

```
BOOL EnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **EnableTrace** method enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace method logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableTrace](#)

CHttpClient::FreezeEvents Method

```
INT FreezeEvents(  
    BOOL bFreeze  
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the client.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableEvents](#), [EnableEvents](#), [RegisterEvent](#)

CHttpClient::GetBearerToken Method

```
INT GetBearerToken(  
    LPTSTR lpszBearerToken,  
    INT nMaxLength  
);  
  
INT GetBearerToken(  
    CString& strBearerToken  
);
```

The **GetBearerToken** method returns the OAuth 2.0 bearer token used to authenticate the client session with a web service.

Parameters

lpszBearerToken

A pointer to a string buffer which will contain the bearer token when the method returns. The string will be null terminated and the buffer must be large enough to accommodate the entire bearer token or the method will fail. This parameter cannot be a NULL pointer. An alternate version of this method accepts a **CString** object.

nMaxLength

The maximum number of characters that may be copied into the string buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the length of the bearer token string. A return value of zero indicates that no bearer token has been specified. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method returns the bearer token which was previously set by a call to **SetBearerToken**. Bearer tokens can be very long strings and do not contain any human readable information. For Google services, these tokens are usually about 180 characters in length. For Microsoft services, their bearer tokens are typically 1,200 characters in length. It is recommended that you provide a buffer size of at least 2,000 characters.

Your application should not store the bearer tokens provided by a web service. These tokens are short-lived and typically only valid for about an hour. If the token has expired, the authorization to access the resource will fail and it must be refreshed. The refresh tokens used to acquire a new bearer token should be stored and they are typically valid for a period of months.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Authenticate](#), [SetBearerToken](#)

CHttpClient::GetCookie Method

```
BOOL GetCookie(  
    LPCTSTR lpszCookieName,  
    LPTSTR lpszCookieValue,  
    INT nCookieValue,  
    LPTSTR lpszCookiePath,  
    INT nCookiePath,  
    LPTSTR lpszCookieDomain,  
    INT nCookieDomain,  
    LPSYSTEMTIME lpCookieExpires,  
    LPDWORD lpdwCookieFlags  
);  
  
BOOL GetCookie(  
    LPCTSTR lpszCookieName,  
    CString& strCookieValue,  
    CString& strCookiePath,  
    CString& lpszCookieDomain,  
    LPSYSTEMTIME lpCookieExpires,  
    LPDWORD lpdwCookieFlags  
);  
  
BOOL GetCookie(  
    LPCTSTR lpszCookieName,  
    CString& strCookieValue  
);
```

The **GetCookie** method returns information about a cookie set by the server.

Parameters

lpszCookieName

A pointer to a string which specifies the name of the cookie to return information about.

lpszCookieValue

A pointer to a string buffer that will contain the value of the cookie. If this information is not required, a NULL pointer may be specified.

nCookieValue

The maximum number of characters that may be copied into the buffer specified by the *lpszCookieValue* parameter, including the terminating null character.

lpszCookiePath

A pointer to a string buffer that will contain the cookie path. If this information is not required, a NULL pointer may be specified.

nCookiePath

The maximum number of characters that may be copied into the buffer specified by the *lpszCookiePath* parameter, including the terminating null character.

lpszCookieDomain

A pointer to a string buffer that will contain the cookie domain. If this information is not required, a NULL pointer may be specified.

nCookieDomain

The maximum number of characters that may be copied into the buffer specified by the

lpzCookieDomain parameter, including the terminating null character.

lpCookieExpires

A pointer to a [SYSTEMTIME](#) structure which specifies the date and time that the cookie expires. If this information is not required, a NULL pointer may be specified.

lpdwCookieFlags

One or more bit flags which specify status information about the cookie. A value of zero indicates that there are no special status flags for the cookie. This parameter may be NULL if the information is not required. The following values are currently defined:

Constant	Description
HTTP_COOKIE_SECURE	This flag specifies that the cookie should only be provided to the server if the connection is secure.
HTTP_COOKIE_SESSION	This flag specifies that the cookie should only be used for the current application session and should not be stored permanently on the local system.

Return Value

If the method succeeds, the return value is non-zero. If the specified cookie does not exist or method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The Hypertext Transfer Protocol uses special tokens called "cookies" to maintain persistent state information between requests for a resource. These cookies are exchanged between the client and server by setting specific header fields. When a server wants the client to use a cookie, it will include a header field named Set-Cookie in the response header when the client requests a resource. The client can then take this cookie and store it, either temporarily in memory or permanently in a file on the local system. The next time that the client requests a resource from that server, it can send the cookie back to the server by setting the Cookie header field. The **GetCookie** method searches for a cookie set by the server in the Set-Cookie header field. The **SetCookie** method creates or modifies the Cookie header field for the next resource requested by the client.

There are two general types of cookies that are used by servers. Session cookies exist only for the duration of the client session; they are stored in memory and not saved in any kind of permanent storage. When the client application terminates, session cookies are deleted and no longer used. Persistent cookies are stored on the local system and are used by the client until their expiration time. It is the responsibility of the client application to store persistent cookies; applications may use a flat text file, a database or any other storage method available.

In addition to the cookie name and value, the server may return additional information about the cookie which the client should use to determine if it should send the cookie back to the server:

The *cookie path* specifies a path for the resources where the cookie should be used. For example, a path of "/" indicates that the cookie should be provided for all resources requested from the server. A path of "/data" would mean that the cookie should be included if the resource is found in the /data folder or a sub-folder, such as /data/projections.asp. However, the cookie would not be provided if the resource /info/status.asp was requested, since it is not in the /data path.

The *cookie domain* specifies the domain for which the cookie should be used. Matches are made by comparing the name of the server against the domain name specified in the

cookie. If the domain is example.com, then any server in the example.com domain would match; for example, both shipping.example.com and orders.example.com would match the domain value. However, if the cookie domain was orders.example.com, then the cookie would only be sent if the resource was requested from orders.example.com, not if the resource was located on shipping.example.com or www.example.com.

The *cookie expiration* specifies the date and time that the cookie should be deleted and no longer sent when a resource is requested from the server. This is only valid for persistent cookies, since session cookies are automatically deleted when the client application terminates. The time is always expressed as Coordinated Universal Time.

The *cookie flags* provide additional information about the cookie. In some cases, a cookie should only be submitted to the server if the resource is requested using a secure connection. In this case, the bit flag HTTP_COOKIE_SECURE will be set.

It is the responsibility of the client application to determine if a cookie meets the criteria required to be submitted to the server. If the application wishes to send the cookie, it can use the **SetCookie** method and specify the cookie name and value.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

See Also

[GetFirstCookie](#), [GetNextCookie](#), [GetHeader](#), [SetCookie](#), [SetHeader](#)

CHttpClient::GetData Method

```
INT GetData(  
    LPCTSTR lpszResource,  
    LPBYTE lpBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwOptions  
);
```

```
INT GetData(  
    LPCTSTR lpszResource,  
    HGLOBAL* lpBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwOptions  
);
```

```
INT GetData(  
    LPCTSTR lpszResource,  
    LPTSTR lpBuffer,  
    DWORD dwMaxLength  
);
```

```
INT GetData(  
    LPCTSTR lpszResource,  
    CString& strBuffer,  
    DWORD dwOptions  
);
```

The **GetData** method requests a resource from the server and copies the data to the specified buffer.

Parameters

lpszResource

A pointer to a string that specifies the resource that will be transferred to the local system. This may be the name of a file on the server, or it may specify the name of a script that will be executed and the output returned to the caller. This string may specify a valid URL for the current server that the client is connected to.

lpBuffer

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the method returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvBuffer* parameter. If the *lpvBuffer* parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual length of the file that was downloaded.

dwMaxLength

An unsigned integer value which specifies the maximum number of characters can be copied into the *lpBuffer* string. This parameter is used with the version of the method that returns the data in a character array and includes the terminating null character. This value must be greater than one and the *lpBuffer* parameter cannot be NULL, otherwise the method will return an error.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_TRANSFER_DEFAULT	The default transfer mode. The resource data is downloaded to the local system exactly as it is stored on the server. If you are requesting a text-based resource, the data may use a different end-of-line character sequence. For example, the end-of-line character may be a single linefeed character instead of a carriage return and linefeed pair.
HTTP_TRANSFER_CONVERT	If the resource being downloaded from the server is textual, the data is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences.
HTTP_TRANSFER_COMPRESS	This option informs the server that the client is willing to accept compressed data. If the server supports compression for the specified resource, then the data will be automatically expanded before being returned to the caller. This option is selected by default if compression has been enabled using the EnableCompression method.
HTTP_TRANSFER_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the method to fail. Instead, the response is returned to the client and the method will succeed. If this option is used, your application should call HttpGetResultCode to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetData** method is used to retrieve a resource from the server and copy it into a local buffer. The method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the *lpBuffer* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpBuffer* parameter point to a global memory handle which will contain the file data when the method returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the

memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. See the example code below.

If the third method or fourth method is used, where the data is returned in a string buffer, the data may be modified so that the end-of-line character sequence matches the convention used by the Windows platform (a carriage return character followed by a linefeed). If Unicode is being used, the data will be converted from a byte array to a Unicode string. An application should only use these versions of the **GetData** method if the resource or remote file contains text.

If compression has been enabled and the server returns compressed data, it will be automatically expanded before being returned to the caller. This will result in a difference between the value returned in the *lpdwLength* parameter, which contains the actual number of bytes copied into the buffer, and the values reported by **GetTransferStatus**. For example, if the server returns 5,000 bytes of compressed data that expands into 15,000 bytes, this function will return 15,000 as the number of bytes copied into the buffer. However, the **GetTransferStatus** method will return the content length as the original 5,000 bytes of compressed data. For this reason, you should always use the value returned in the *lpdwLength* parameter to determine the amount of data that has been copied into the buffer.

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Example

```
HGLOBAL hglobalBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Return the file data into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hglobalBuffer parameter
nResult = pClient->GetData(lpszResource, &hglobalBuffer, &cbBuffer);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // resource data
    lpBuffer = (LPBYTE)GlobalLock(hglobalBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hglobalBuffer);
    GlobalFree(hglobalBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttp10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableCompression](#), [EnableEvents](#), [GetFile](#), [GetText](#), [GetTransferStatus](#), [PostData](#), [PutData](#), [PutFile](#), [RegisterEvent](#)

CHttpClient::GetEncodingType Method

```
INT GetEncodingType();
```

The **GetEncodingType** method determines which content-encoding option is enabled.

Parameters

None.

Return Value

If the method succeeds, the return value is one of the following values:

Value	Constant	Description
0	HTTP_ENCODING_NONE	No encoding will be applied to the content of a request, and no Content-Type header will be generated.
1	HTTP_ENCODING_URL	URL encoding will be applied to the content of a request, and a Content-Type header will be generated with the value "application/x-www-form-urlencoded"
2	HTTP_ENCODING_XML	URL encoding will be applied to the content of a request, EXCEPT that spaces will not be replaced by '+'. This encoding type is intended for use with XML parsers that do not recognize '+' as a space. A Content-Type header will be generated with the value "application/x-www-form-urlencoded".

If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Command](#), [PostData](#), [SetEncodingType](#)

CHttpClient::GetErrorString Method

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);  
  
INT GetErrorString(  
    DWORD dwErrorCode,  
    CString& strDescription  
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLastError](#), [SetLastError](#), [ShowError](#)

CHttpClient::GetFile Method

```
INT GetFile(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszRemoteFile,  
    DWORD dwOptions  
);
```

The **GetFile** method downloads the specified file on the server to the local system.

Parameters

lpszLocalFile

A pointer to a string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A pointer to a string that specifies the resource on the server. This may be the name of a file on the server, or it may specify the name of a script that will be executed and the output returned to the caller. This string may specify a valid URL for the current server that the client is connected to.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_TRANSFER_DEFAULT	The default transfer mode. The resource data is downloaded to the local system exactly as it is stored on the server. If you are requesting a text-based resource, the data may use a different end-of-line character sequence. For example, the end-of-line character may be a single linefeed character instead of a carriage return and linefeed pair.
HTTP_TRANSFER_CONVERT	If the resource being downloaded from the server is textual, the data is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences.
HTTP_TRANSFER_COMPRESS	This option informs the server that the client is willing to accept compressed data. If the server supports compression for the specified resource, then the data will be automatically expanded before being returned to the caller. This option is selected by default if compression has been enabled using the EnableCompression method.
HTTP_TRANSFER_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the method to fail. Instead, the response is returned to the

client and the method will succeed. If this option is used, your application should call **HttpGetResultCode** to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

Enabling compression does not guarantee that the data returned by the server will actually be compressed, it only informs the server that the client is willing to accept compressed data. Whether or not a particular resource is compressed depends on the server configuration, and the server may decide to only compress certain types of resources, such as text files. To determine if the server compressed the data returned to the client, use the **GetHeader** method to get the value of the **Content-Encoding** header after this function returns. If the header is defined, the value specifies the compression method used, otherwise the data was not compressed.

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableCompression](#), [EnableEvents](#), [GetData](#), [GetText](#), [GetTransferStatus](#), [PostData](#), [PutData](#), [PutFile](#), [RegisterEvent](#)

CHttpClient::GetFileSize Method

```
INT GetFileSize(  
    LPCTSTR lpszFileName,  
    LPDWORD lpdwFileSize  
);
```

The **GetFileSize** method returns the size of the specified file on the server.

Parameters

lpszFileName

Points to a string that specifies the name of the remote file.

lpdwFileSize

Points to an unsigned integer that will contain the size of the specified file in bytes.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method uses the HEAD command to retrieve header information about the file without downloading the contents of the file itself. This requires that the server support at least version 1.0 of the protocol standard, or an error will be returned.

The server may not return a file size for some resources. This is typically the case with scripts that generate dynamic content because the server has no way of determining the size of the output generated by the script without actually executing it. The server may also not provide a file size for HTML documents which use server side includes (SSI) because that content is also dynamically created by the server. If the request to the server was successful and the file exists, but the server does not return a file size, the method will succeed but the file size returned to the caller will be zero.

When a request is made to the server for information about the file, the library will attempt to keep the connection alive, even if the HTTP_OPTION_KEEPALIVE option has not been specified for the session. This allows an application to request the file size and then download the file without having to write additional code to re-establish the connection. However, it is possible that the attempt to keep the connection open will fail. In that case, an error will be returned and the client handle will no longer be valid. If this happens, the *lpdwFileSize* parameter may still contain a valid value. If the library was able to determine the file size, but was not able to maintain the connection to the server, the returned file size will be greater than zero even if the method returns an error.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttp10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Command](#), [GetFileTime](#), [VerifyFile](#)

CHttpClient::GetFileTime Method

```
INT GetFileTime(  
    LPCTSTR lpzFileName,  
    LPSYSTEMTIME lpFileTime,  
    BOOL bLocalize  
);
```

The **GetFileTime** method returns the modification time for the specified file on the server.

Parameters

lpzFileName

Points to a string that specifies the name of the remote file.

lpFileTime

Points to a [SYSTEMTIME](#) structure that will be set to the current modification time for the remote file.

bLocalize

A boolean flag which specifies if the file time is localized to the current timezone. If this value is non-zero, then the file time is adjusted to that the time is local to the current system. If this value is zero, the file time is returned in UTC time.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetFileTime** method can be used to determine the date and time that a file was last modified on the server. The time may either be localized to the current system, or it may be returned as UTC time.

This method uses the HEAD command to retrieve header information about the file without downloading the contents of the file itself. This requires that the server support at least version 1.0 of the protocol standard, or an error will be returned.

The server may not return a modification time for some resources. If the request to the server was successful and the file exists, but the server does not return a modification time, the method will succeed but all of the members of the SYSTEMTIME structure will be zero.

When a request is made to the server for information about the file, the library will attempt to keep the connection alive, even if the HTTP_OPTION_KEEPALIVE option has not been specified for the session. This allows an application to request the modification time and then download the file without having to write additional code to re-establish the connection. However, it is possible that the attempt to keep the connection open will fail. In that case, an error will be returned and the client handle will no longer be valid. If this happens, the SYSTEMTIME structure may still contain a valid value. If the library was able to determine the modification time, but was not able to maintain the connection to the server, the members of the SYSTEMTIME structure will specify a valid date and time.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Command](#), [GetFileSize](#)

CHttpClient::GetFirstCookie Method

```
BOOL GetFirstCookie(  
    LPTSTR lpszCookieName,  
    LPINT lpnNameLen,  
    LPTSTR lpszCookieValue,  
    LPINT lpnValueLen  
);  
  
BOOL GetFirstCookie(  
    CString& strCookieName,  
    CString& lpszCookieValue  
);
```

The **GetFirstCookie** method returns the first cookie set by the server.

Parameters

lpszCookieName

A pointer to a string buffer which will contain the name of the first cookie set by the server.

lpnNameLen

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the method returns, this value is updated to specify the actual length of the cookie name string.

lpszCookieValue

A pointer to a string buffer which will contain the name of the first cookie value set by the server. If the cookie value is not required, this parameter may be NULL.

lpnValueLen

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the method returns, this value is updated to specify the actual length of the cookie value string. If the *lpszCookieValue* parameter is NULL, this parameter should also be NULL.

Return Value

If the method succeeds, the return value is non-zero. If there are no cookies or the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetFirstCookie** method is used to enumerate the cookies set by the server after a resource has been requested. To get information about a specific cookie, use the **GetCookie** method.

Example

```
CString strCookieName;  
CString strCookieValue;  
BOOL bResult;  
  
// Get the first cookie set by the server  
bResult = pClient->GetFirstCookie(strCookieName, strCookieValue);  
  
while (bResult)  
{  
    // The strCookieName and strCookieValue strings contain the
```

```
// the name and value for a cookie set by the server

// Get the next cookie set by the server
bResult = pClient->GetNextCookie(strCookieName, strCookieValue);
};
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

See Also

[GetCookie](#), [GetNextCookie](#), [GetHeader](#), [SetCookie](#), [SetHeader](#)

CHttpClient::GetFirstHeader Method

```
BOOL GetFirstHeader(  
    LPTSTR lpszHeader,  
    LPINT lpcchHeader  
    LPTSTR lpszValue,  
    LPINT lpcchValue  
);  
  
BOOL GetFirstHeader(  
    CString& strHeader,  
    CString& strValue  
);
```

The **GetFirstHeader** method returns the name and value of the first response header.

Parameters

lpszHeader

A pointer to a string buffer that will contain the name of the header field when the method returns.

lpcchHeader

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the method returns, this value is updated to specify the actual length of the header name string.

lpszValue

A pointer to a string buffer that will contain the name of the header value when the method returns.

lpcchValue

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the method returns, this value is updated to specify the actual length of the header value string.

Return Value

If the method succeeds, the return value is non-zero. If the header field does not exist or the client handle is invalid, the method returns a value of zero. To get extended error information, call **GetLastError**.

Remarks

Use this method together with **GetNextHeader** to enumerate all request or response headers.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHeader](#), [GetNextHeader](#), [SetHeader](#)

CHttpClient::GetFormProperties Method

```
INT GetFormProperties(  
    LPHTTPFORMPROPERTIES LpFormProp  
);
```

The **GetFormProperties** function returns information about the current form.

Parameters

LpFormProp

Points to a HTTPFORMPROPERTIES structure which will contain information about the current form.

Return Value

If the method succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateForm](#), [SetFormProperties](#), [SubmitForm](#), [HTTPFORMPROPERTIES](#)

CHttpClient::GetHandle Method

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshttpv10.lib`

See Also

[AttachHandle](#), [DetachHandle](#), [IsInitialized](#)

CHttpClient::GetHeader Method

```
BOOL GetHeader(  
    LPCTSTR lpszHeader,  
    LPTSTR lpszValue,  
    INT nMaxLength  
);
```

```
BOOL GetHeader(  
    LPCTSTR lpszHeader,  
    CString& strValue  
);
```

The **GetHeader** method returns the value of the specified response header field.

Parameters

lpszHeader

Pointer to a string which specifies the header value to be returned.

lpszValue

Pointer to a buffer which will contain the null-terminated string value of the specified header field. This may also be a **CString** object which will contain the value of the header field when the function returns.

nMaxLength

Maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is non-zero. If the header field does not exist or the client handle is invalid, the method returns a value of zero. To get extended error information, call **GetLastError**.

Remarks

When a resource is returned by the server, it consists of three parts. The first part consists of a single line that indicates the result of the request. The second part is one or more header fields which provides specific information about the resource, such as its size in bytes. The third part consists of the resource data itself, such as the HTML document or image data. For example, this is what the response to a request for a simple HTML document can look like:

```
HTTP/1.0 200 OK  
Date: Mon, 5 Jan 2004 20:18:33 GMT  
Content-Type: text/html  
Last-Modified: Mon, 5 Jan 2004 19:34:19 GMT  
Content-Length: 115
```

```
<html>  
<head>  
<title>Simple Document</title>  
</head>  
<body>  
This is a simple HTML document.  
</body>  
</html>
```

The first line consists of the protocol version, a numeric response code and some text describing

the result. The subsequent lines are the header, which is similar to the headers used in email messages. For example, the Date field specifies the date the resource was requested, the Content-Type field specifies what type of resource was requested, and the Content-Length field specifies the size of the resource in bytes. The end of the header block is indicated by an empty line (two carriage-return/linefeed sequences), and is followed by the resource itself, in this case a simple HTML document.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetCookie](#), [GetFirstCookie](#), [GetFirstHeader](#), [GetNextCookie](#), [GetNextHeader](#), [SetHeader](#)

CHttpClient::GetLastError Method

```
DWORD GetLastError();  
  
DWORD GetLastError(  
    CString& strDescription  
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError(0)** when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or HTTP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

See Also

[GetErrorString](#), [SetLastError](#), [ShowError](#)

CHttpClient::GetFirstCookie Method

```
DWORD GetNextCookie(  
    LPTSTR lpszCookieName,  
    LPINT lpnNameLen,  
    LPTSTR lpszCookieValue,  
    LPINT lpnValueLen  
);
```

```
DWORD GetNextCookie(  
    CString& strCookieName,  
    CString& strCookieValue  
);
```

The **GetNextCookie** method returns the next cookie set by the server.

Parameters

lpszCookieName

A pointer to a string buffer which will contain the name of the first cookie set by the server.

lpnNameLen

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the method returns, this value is updated to specify the actual length of the cookie name string.

lpszCookieValue

A pointer to a string buffer which will contain the name of the first cookie value set by the server. If the cookie value is not required, this parameter may be NULL.

lpnValueLen

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the method returns, this value is updated to specify the actual length of the cookie value string. If the *lpszCookieValue* parameter is NULL, this parameter should also be NULL.

Return Value

If the method succeeds, the return value is non-zero. If there are no more cookies or the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetNextCookie** method is used to enumerate the cookies set by the server after a resource has been requested. To get information about a specific cookie, use the **GetCookie** method.

Example

```
CString strCookieName;  
CString strCookieValue;  
BOOL bResult;  
  
// Get the first cookie set by the server  
bResult = pClient->GetFirstCookie(strCookieName, strCookieValue);  
  
while (bResult)  
{  
    // The strCookieName and strCookieValue strings contain the  
    // the name and value for a cookie set by the server
```

```
    // Get the next cookie set by the server
    bResult = pClient->GetNextCookie(strCookieName, strCookieValue);
};
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

See Also

[GetCookie](#), [GetFirstCookie](#), [GetHeader](#), [SetCookie](#), [SetHeader](#)

CHttpClient::GetNextHeader Method

```
BOOL GetNextHeader(  
    LPTSTR lpszHeader,  
    LPINT lpcchHeader,  
    LPTSTR lpszValue,  
    LPINT lpcchValue  
);  
  
BOOL GetNextHeader(  
    CString& strHeader,  
    CString& strValue  
);
```

The **GetNextHeader** method returns the name and value of the next response header.

Parameters

lpszHeader

A pointer to a string buffer that will contain the name of the header field when the method returns.

lpcchHeader

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the method returns, this value is updated to specify the actual length of the header name string.

lpszValue

A pointer to a string buffer that will contain the name of the header value when the method returns.

lpcchValue

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the method returns, this value is updated to specify the actual length of the header value string.

Return Value

If the method succeeds, the return value is non-zero. If the header field does not exist or the client handle is invalid, the method returns a value of zero. To get extended error information, call **GetLastError**.

Remarks

Use this method iteratively after **GetFirstHeader** to enumerate all request or response headers.

Unlike the **GetHeader** method, which returns a single header name and value, the **GetFirstHeader** and **GetNextHeader** methods will return multiple headers that have the same common name.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHeader](#), [GetFirstHeader](#), [SetHeader](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CHttpClient::GetOption Method

```
INT GetOption(  
    DWORD dwOption,  
    LPBOOL LpbEnabled  
);
```

The **GetOption** method determines whether a specified HTTP option is enabled.

Parameters

dwOption

An unsigned integer which specifies the option that is to be checked. It may be one of the following values:

Constant	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_ERRORDATA	This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.
HTTP_OPTION_NOUSERAGENT	This option specifies the client should not include a User-Agent header with any requests made during the session. The user agent is a string which is used to identify the client application to the server.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and

	remote port number, default capability selection and how the connection is established.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol. The client will default to using TLS 1.2 or later for secure connections.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
HTTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

lpbEnabled

A pointer to an integer which will be set to a non-zero value when the method returns if the specified option has been enabled. If the option has not been enabled, a zero value will be returned in the variable.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**. Note that the value returned

in the *lpbEnabled* parameter is only valid if the method succeeds.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

See Also

[Connect](#), [SetOption](#)

CHttpClient::GetPriority Method

```
INT GetPriority();
```

The **GetPriority** method returns a value which specifies the priority of file transfers.

Parameters

None.

Return Value

If the method succeeds, the return value is the current file transfer priority. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetPriority** method can be used to determine the current priority assigned to file transfers performed by the client. It may be one of the following values:

Constant	Description
HTTP_PRIORITY_NORMAL	The default priority which balances resource utilization and transfer speed. It is recommended that most applications use this priority.
HTTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the transfer. It is typically used with worker threads running in the background when the amount of time required perform the transfer is not critical.
HTTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the transfer and meters the bandwidth allocated for the transfer. This priority will increase the average amount of time required to complete a file transfer.
HTTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the transfer, allocating more memory for internal buffering. It can be used when it is important to transfer the file quickly, and there are no other threads currently performing file transfers at the time.
HTTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization while attempting to transfer the file as quickly as possible. If the file transfer is being performed in the main UI thread, this priority can cause the application to appear to become non-responsive. No events will be generated during the transfer.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

CHttpClient::GetResultCode Method

```
INT GetResultCode();
```

The **GetResultCode** method reads the result code returned by the server in response to a command. The result code is a three-digit numeric code, and indicates if the operation succeeded, failed or requires additional action by the client.

Parameters

None.

Return Value

If the method succeeds, the return value is the result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

Result codes are three-digit numeric values returned by the server. They may be broken down into the following ranges:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

See Also

[Command](#), [GetResultString](#)

CHttpClient::GetResultString Method

```
INT GetResultString(  
    LPTSTR lpszResult,  
    INT cbResult  
);
```

```
INT GetResultString(  
    CString& strResult  
);
```

The **GetResultString** method returns the last message sent by the server along with the result code.

Parameters

lpszResult

A pointer to the buffer that will contain the result string returned by the server. An alternate form of the method accepts a **CString** argument which will contain the result string returned by the server.

cbResult

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetResultString** method is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **GetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Command](#), [GetResultCode](#)

CHttpClient::GetSecurityInformation Method

```
BOOL GetSecurityInformation(  
    LPSECURITYINFO LpSecurityInfo  
);
```

The **GetSecurityInformation** method returns security protocol, encryption and certificate information about the current client connection.

Parameters

LpSecurityInfo

A pointer to a [SECURITYINFO](#) structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;  
securityInfo.dwSize = sizeof(SECURITYINFO);  
  
if (pClient->GetSecurityInformation(&securityInfo))  
{  
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)  
    {  
        MessageBox(NULL, _T("The connection is not secure"),  
                    _T("Connection"), MB_OK);  
    }  
    else  
    {  
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)  
        {  
            MessageBox(NULL, _T("The connection is secure"),  
                        _T("Connection"), MB_OK);  
        }  
        else  
        {  
            MessageBox(NULL, _T("The server certificate not valid"),  
                        _T("Connection"), MB_OK);  
        }  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [CreateSecurityCredentials](#), [SECURITYINFO](#)

CHttpClient::GetStatus Method

```
INT GetStatus();
```

The **GetStatus** method the current status of the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the client status code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetStatus** method returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Constant	Description
0	HTTP_STATUS_UNUSED	No connection has been established.
1	HTTP_STATUS_IDLE	The client is current idle and not sending or receiving data.
2	HTTP_STATUS_CONNECT	The client is establishing a connection with the server.
3	HTTP_STATUS_READ	The client is reading data from the server.
4	HTTP_STATUS_WRITE	The client is writing data to the server.
5	HTTP_STATUS_DISCONNECT	The client is disconnecting from the server.
6	HTTP_STATUS_GETDATA	The client is downloading data from the server.
7	HTTP_STATUS_PUTDATA	The client is uploading data to the server.
8	HTTP_STATUS_POSTDATA	The client is posting data to a script on the server.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session. To obtain status information about a file transfer, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

See Also

[IsBlocking](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#), [GetTransferStatus](#)

CHttpClient::GetText Method

```
INT GetText(  
    LPCTSTR lpszResource,  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);
```

The **GetText** method copies the contents of a text file on the server, or the text output from a script, to the specified string buffer.

Parameters

lpszResource

A pointer to a string that specifies the resource that will be transferred to the local system. This may be the name of a file on the server, or it may specify the name of a script that will be executed and the output returned to the caller. This string may specify a valid URL for the current server that the client is connected to.

lpszBuffer

A pointer to a string buffer which will contain the contents of the text file when the method returns. This buffer should be large enough to store the contents of the file, including a terminating null character. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. This value must be larger than zero. If this value is smaller than the actual size of the text file, the data returned will be truncated.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetText** method is used to download a text file or retrieve the text output from a script and store the contents in a string buffer. Because binary data can include embedded null characters which would truncate the string, this method should only be used with text files or script output that is known to be textual. For example, it is safe to use this method when a resource returns HTML or XML data, but should not be used if it returns an image or executable file.

This method has been included as a convenience for applications that need to retrieve relatively small amounts of textual data and manipulate the contents as a string. If the Unicode version of this method is called, the text is automatically converted to a Unicode string. If the maximum amount of data being returned is unknown or the amount of text is very large, it is recommended that you use the **GetData** or **GetFile** methods.

If you use the **GetFileSize** method to determine how large the string buffer should be prior to calling this method, it is important to be aware that the actual number of characters may differ based on the end-of-line conventions used by the host operating system. For example, if you call **GetFileSize** to obtain the size of a text file on a UNIX system, the value will not be large enough to store the complete file because UNIX uses a single linefeed (LF) character to indicate the end-of-line, while a Windows system will use a carriage-return and linefeed (CRLF) pair. To accommodate this difference, you should always allocate extra memory for the string buffer to store the additional end-of-line characters.

HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Example

```
LPTSTR lpszBuffer = (LPTSTR)calloc(MAXFILESIZE, sizeof(TCHAR));

if (lpszBuffer == NULL)
    return;

nResult = pHttpClient->GetText(lpszRemoteFile, lpszBuffer, MAXFILESIZE);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableEvents](#), [GetData](#), [GetFile](#), [GetTransferStatus](#), [PutData](#), [PutFile](#), [RegisterEvent](#)

CHttpClient::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

See Also

[Connect](#), [IsReadable](#), [IsWritable](#), [Read](#), [SetTimeout](#), [Write](#)

CHttpClient::GetTransferStatus Method

```
INT GetTransferStatus(  
    LPHTTPTRANSFERSTATUS lpStatus  
);  
  
INT GetTransferStatus(  
    LPHTTPTRANSFERSTATUSEX lpStatus  
);
```

The **GetTransferStatus** method returns information about the current data transfer in progress.

Parameters

lpStatus

A pointer to an **HTTPTRANSFERSTATUS** or **HTTPTRANSFERSTATUSEX** structure which contains information about the status of the current data transfer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is **HTTP_ERROR**. To get extended error information, call **GetLastError**.

Remarks

The **GetTransferStatus** method returns information about the current data transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no data currently being transferred, this method will return the status of the last successful data transfer made by the client.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshttpv10.lib`

See Also

[EnableEvents](#), [GetStatus](#), [RegisterEvent](#), [HTTPTRANSFERSTATUS](#), [HTTPTRANSFERSTATUSEX](#)

CHttpClient::IsBlocking Method

BOOL IsBlocking();

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking method may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Cancel](#), [GetStatus](#), [IsConnected](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#), [Read](#), [Write](#)

CHttpClient::IsConnected Method

```
BOOL IsConnected();
```

The **IsConnected** method is used to determine if the client is currently connected to a server.

Parameters

None.

Return Value

If the client is connected to a server, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#)

CHttpClient::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

See Also

[CHttpClient](#), [IsBlocking](#), [IsConnected](#)

CHttpClient::IsReadable Method

```
BOOL IsReadable(  
    INT nTimeout,  
    LPDWORD lpdwAvail  
);
```

The **IsReadable** method is used to determine if data is available to be read from the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the method returns a non-zero value. If the client cannot read any data, the method returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsConnected](#), [IsInitialized](#), [IsWritable](#), [Write](#)

CHttpClient::IsWritable Method

```
BOOL IsWritable(  
    INT nTimeout  
);
```

The **IsWritable** method is used to determine if data can be written to the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the method returns a non-zero value. If the client cannot write any data, the method returns zero.

Remarks

Although this method can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: cshttpv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsConnected](#), [IsInitialized](#), [IsReadable](#), [Write](#)

CHttpClient::OpenFile Method

```
INT OpenFile(  
    LPCTSTR lpszFileName  
);
```

The **OpenFile** method opens the specified file on the server.

Parameters

lpszFileName

Points to a string that specifies the name of the remote file to open.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

Only one file may be opened at a time for each client session. Use the **CloseFile** method to close the file on the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CloseFile](#)

CHttpClient::PatchData Method

```
INT PatchData(  
    LPCTSTR lpszResource,  
    LPCTSTR lpszPatchData,  
    LPBYTE lpResult,  
    LPDWORD lpcbResult,  
    DWORD dwOptions  
);
```

```
INT PatchData(  
    LPCTSTR lpszResource,  
    LPCTSTR lpszPatchData,  
    HGLOBAL* lpResult,  
    LPDWORD lpcbResult,  
    DWORD dwOptions  
);
```

```
INT PatchData(  
    LPCTSTR lpszResource,  
    LPCTSTR lpszPatchData,  
    CString& strResult,  
    DWORD dwOptions  
);
```

The **PatchData** method submits patch data to the server and returns the result in a buffer provided by the caller.

Parameters

hClient

Handle to the client session.

lpszResource

A pointer to a string that specifies the resource name that the patch data will be submitted to. Typically this is the name of a script on the server.

lpszPatchData

A pointer to a string that specifies the patch data that will be submitted to the server.

lpvResult

A pointer to a byte buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the method returns. An alternate version of this method accepts a **CString** object which will contain the server response.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvResult* parameter. If the *lpvResult* parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual number of bytes of data that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
----------	-------------

HTTP_PATCH_DEFAULT	The default patch mode. The contents of the buffer will be submitted without encoding. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
HTTP_PATCH_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **PatchData** method is used to submit XML or JSON formatted patch data to a service, and then returns a copy of the response from the server into a local buffer. This method will not perform any encoding and will not automatically define the type of patch data being submitted. Your application is responsible for specifying the content type for the patch data, and ensuring that the XML or JSON data that is being submitted to the server is formatted correctly.

This method sends a PATCH command to the server, which is similar to a POST or PUT request. It is used to make partial updates to a resource, rather than creating or replacing it entirely. The format of the patch data is specific to the service being used. If the resource being patched does not exist, the behavior is defined by the server. If enough information is provided, it may choose to create the resource just as if a PUT command was used, or it may return an error.

Your application should use the **SetHeader** method to define the Content-Type header prior to calling the **PatchData** method. One of the most common formats used is the JSON Merge Patch which is defined in RFC 7396. The value for the Content-Type header for this patch format is "application/merge-patch+json". Refer to your service API documentation to determine what patch formats are acceptable, along with any additional header values that must be defined.

The method may be used in one of two ways, depending on the needs of the application. The first approach is to pre-allocate a buffer large enough to store the resulting output of the script. In this case, the *lpvResult* parameter will point to the buffer that was allocated by the client and the value that the *lpcbResult* parameter points to should be initialized to the size of that buffer.

The second approach is to have the *lpvResult* parameter point to a global memory handle which will contain the data when the method returns. In this case, the value that the *lpcbResult* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. See the example code below.

This method will cause the current thread to block until the post completes, a timeout occurs or the operation is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of the transaction while it is in progress, use the `GetTransferStatus` method.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Store the script output into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter. Since the output from the
// script is textual, the HTTP_PATCH_CONVERT option is used

nResult = lpClient->PatchData(lpszResource,
                             lpszPatchData,
                             &hgblBuffer,
                             &cbBuffer,
                             HTTP_PATCH_CONVERT);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshttpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableEvents](#), [GetData](#), [GetTransferStatus](#), [PostData](#), [PostJson](#), [PostXml](#), [RegisterEvent](#)

CHttpClient::PostData Method

```
INT PostData(  
    LPCTSTR lpszResource,  
    LPBYTE lpBuffer,  
    DWORD cbBuffer,  
    LPBYTE lpResult,  
    LPDWORD lpcbResult,  
    DWORD dwOptions  
);
```

```
INT PostData(  
    LPCTSTR lpszResource,  
    LPBYTE lpBuffer,  
    DWORD cbBuffer,  
    HGLOBAL* lpResult,  
    LPDWORD lpcbResult,  
    DWORD dwOptions  
);
```

```
INT PostData(  
    LPCTSTR lpszResource,  
    LPCTSTR lpszBuffer,  
    CString& strResult,  
    DWORD dwOptions  
);
```

The **PostData** method submits the contents of the specified buffer to a script on the server and returns the result in a buffer provided by the caller.

Parameters

lpszResource

A pointer to a string that specifies the resource that the data will be posted to on the server. Typically this is the name of an executable script. This string may specify a valid URL for the current server that the client is connected to.

lpBuffer

A pointer to the data that will be provided to the script. This parameter may be NULL if the script does not require any additional data from the client. In an alternate form of the method, this may point to a string which contains the data to be posted.

cbBuffer

The number of bytes to copy from the buffer. If the *lpBuffer* parameter is NULL, this value should be zero.

lpResult

A pointer to a byte buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the method returns. An alternate version of this method accepts a **CString** object which will contain the server response.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvBuffer* parameter. If the *lpvResult* parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual number of bytes of data

that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_POST_DEFAULT	The default post mode. The contents of the buffer are encoded and sent as standard form data. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
HTTP_POST_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.
HTTP_POST_MULTIPART	The contents of the buffer being sent to the server consists of multipart form data and will be sent as-is without any encoding.
HTTP_POST_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the method to fail. Instead, the response is returned to the client and the method will succeed.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **PostData** method is used to submit data to a script that executes on the server and then copy the output from that script into a local buffer. If you are submitting XML formatted data to the server, it is recommended that you use the **PostXml** method to ensure that the correct content type and encoding is automatically selected.

The method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the resulting output of the script. In this case, the *lpvResult* parameter will point to the buffer that was allocated by the client and the value that the *lpcbResult* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvResult* parameter point to a global memory handle which will contain the data when the method returns. In this case, the value that the *lpcbResult* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. See the example code below.

It is common for servers to return additional information about a failed request in their response to the client. If you need to process this information, use the HTTP_POST_ERRORDATA option

which causes the error message to be returned in the *lpvResult* buffer provided by the caller. If this option is used, your application should call **GetResultCode** to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This method will cause the current thread to block until the post completes, a timeout occurs or the operation is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Store the script output into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter. Since the output from the
// script is textual, the HTTP_POST_CONVERT option is used

nResult = lpClient->PostData(lpszResource,
                             lpParameters,
                             cbParameters,
                             &hgblBuffer,
                             &cbBuffer,
                             HTTP_POST_CONVERT);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableEvents](#), [GetData](#), [GetTransferStatus](#), [PostJson](#), [PostXml](#), [PutData](#), [RegisterEvent](#)

CHttpClient::PostFile Method

```
INT PostFile(  
    LPCTSTR lpszFileName,  
    LPCTSTR lpszResource,  
    LPCTSTR lpszFieldName  
);
```

The **PostFile** method posts the contents of the specified file to a script executed on the server.

Parameters

lpszFileName

A pointer to a string that specifies the file that will be transferred from the local system. The file pathing and name conventions must be that of the local host.

lpszResource

A pointer to a string that specifies the resource on the server that the file data will be posted to. Typically this is the name of a script that is responsible for processing and storing the file data.

lpszFieldName

A pointer to a string that corresponds to the form field name that the script expects. If this parameter is NULL or an empty string, a default field name of "File1" is used.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **PostFile** method is similar to the **PutFile** method in that it can be used to upload the contents of a local file to a server. However, instead of using the PUT command, the POST command is used to send the file data to a script that is executed on the server. This method has the advantage of not requiring any special configuration settings on the server, however it does require that the script be able to process multipart/form-data as defined in RFC 2388.

To support uploading files from a form on a webpage, the FILE input type is used along with the action that specifies the script that will accept the file data and process it. For example, the HTML code could look like this:

```
<form action="/cgi-bin/upload.cgi" method="post" enctype="multipart/form-data">  
<input type="file" name="datafile" size="20">  
<input type="submit">  
</form>
```

In this example, the script /cgi-bin/upload.cgi is responsible for processing the file data that is posted by the client, and the form field name "datafile" is used. The user can select a file, and when the Submit button is clicked, the file data is posted to the script. To simulate this using the **PostFile** method, the *lpszFileName* parameter should be set to the name of the local file that will be posted to the server. The *lpszResource* parameter should be the name of the script, in this case "/cgi-bin/upload.cgi". The *lpszFieldName* parameter should be specified as the string "datafile" to match the name of the field used by the form.

Note that the **PostFile** method always submits the file contents as multipart/form-data with the content type set to application/octet-stream. The script that accepts the posted data must be able to parse the multipart header block and correctly process 8-bit data. If the script assumes that the

data will be posted using a specific encoding type such as base64 then the file data may not be accepted or may be corrupted by the script.

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableEvents](#), [GetData](#), [GetFile](#), [GetTransferStatus](#), [PostData](#), [PutData](#), [PutFile](#), [RegisterEvent](#)

CHttpClient::PostJson Method

```
INT PostJson(  
    LPCTSTR lpszResource,  
    LPCTSTR lpszJsonData,  
    LPBYTE lpResult,  
    LPDWORD lpcbResult,  
    DWORD dwOptions  
);
```

```
INT PostJson(  
    LPCTSTR lpszResource,  
    LPCTSTR lpszJsonData,  
    HGLOBAL* lpResult,  
    LPDWORD lpcbResult,  
    DWORD dwOptions  
);
```

```
INT PostJson(  
    LPCTSTR lpszResource,  
    LPCTSTR lpszJsonData,  
    CString& strResult,  
    DWORD dwOptions  
);
```

The **PostJson** method submits JSON formatted data to a script on the server and returns the result in a buffer provided by the caller.

Parameters

lpszResource

A pointer to a string that specifies the resource that the data will be posted to on the server. Typically this is the name of an executable script.

lpszJsonData

A pointer to a string that specifies the JSON data that will be submitted to the server.

lpResult

A pointer to a byte buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the method returns. An alternate version of this method accepts a **CString** object which will contain the server response.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvResult* parameter. If the *lpvResult* parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual number of bytes of data that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_POST_DEFAULT	The default post mode. The contents of the buffer will be submitted without encoding and should use the standard

	JSON format. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
HTTP_POST_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.
HTTP_POST_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the method to fail. Instead, the response is returned to the client and the method will succeed.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **PostJson** method is used to submit JSON formatted data to a script that executes on the server and then copy the output from that script into a local buffer. This function automatically sets the correct content type and encoding required for submitting JSON data to a server, however it does not parse the JSON data itself to ensure that it is well-formed. Your application is responsible for ensuring that the JSON data that is being submitted to the server is formatted correctly.

The method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the resulting output of the script. In this case, the *lpvResult* parameter will point to the buffer that was allocated by the client and the value that the *lpcbResult* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvResult* parameter point to a global memory handle which will contain the data when the method returns. In this case, the value that the *lpcbResult* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. See the example code below.

If the content type for the request has not been explicitly specified, it will be automatically updated by this function to use the "application/json" content type. You can override the default content type for the request by calling the **SetHeader** method prior to calling this method. Most servers require you to explicitly specify what type of data is being submitted by the client and will reject requests which do not correctly identify the content type.

It is common for servers to return additional information about a failed request in their response to the client. If you need to process this information, use the HTTP_POST_ERRORDATA option which causes the error message to be returned in the *lpvResult* buffer provided by the caller. If this option is used, your application should call **GetResultCode** to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This method will cause the current thread to block until the post completes, a timeout occurs or

the operation is canceled. During the transfer, the `HTTP_EVENT_PROGRESS` event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Store the script output into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter. Since the output from the
// script is textual, the HTTP_POST_CONVERT option is used
nResult = lpClient->PostJson(lpszResource,
                             lpszJsonData,
                             &hgblBuffer,
                             &cbBuffer,
                             HTTP_POST_CONVERT);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshttpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableEvents](#), [GetData](#), [GetTransferStatus](#), [PostData](#), [PostXml](#), [PutData](#), [RegisterEvent](#)

CHttpClient::PostXml Method

```
INT PostXml(  
    LPCTSTR lpszResource,  
    LPCTSTR lpszXmlData,  
    LPBYTE lpResult,  
    LPDWORD lpcbResult,  
    DWORD dwOptions  
);
```

```
INT PostXml(  
    LPCTSTR lpszResource,  
    LPCTSTR lpszXmlData,  
    HGLOBAL* lpResult,  
    LPDWORD lpcbResult,  
    DWORD dwOptions  
);
```

```
INT PostXml(  
    LPCTSTR lpszResource,  
    LPCTSTR lpszXmlData,  
    CString& strResult,  
    DWORD dwOptions  
);
```

The **PostXml** method submits XML formatted data to a script on the server and returns the result in a buffer provided by the caller.

Parameters

lpszResource

A pointer to a string that specifies the resource that the data will be posted to on the server. Typically this is the name of an executable script.

lpszXmlData

A pointer to a string that specifies the XML data that will be submitted to the server.

lpResult

A pointer to a byte buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the method returns. An alternate version of this method accepts a **CString** object which will contain the server response.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvResult* parameter. If the *lpvResult* parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual number of bytes of data that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_POST_DEFAULT	The default post mode. The contents of the buffer will be submitted without encoding and should use the standard

	XML format. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
HTTP_POST_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.
HTTP_POST_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the method to fail. Instead, the response is returned to the client and the method will succeed.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **PostXml** method is used to submit XML formatted data to a script that executes on the server and then copy the output from that script into a local buffer. This function automatically sets the correct content type and encoding required for submitting XML data to a server, however it does not parse the XML data itself to ensure that it is well-formed. Your application is responsible for ensuring that the XML data that is being submitted to the server is formatted correctly.

The method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the resulting output of the script. In this case, the *lpvResult* parameter will point to the buffer that was allocated by the client and the value that the *lpcbResult* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvResult* parameter point to a global memory handle which will contain the data when the method returns. In this case, the value that the *lpcbResult* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. See the example code below.

If the content type for the request has not been explicitly specified, it will be automatically updated by this function to use the "text/xml" content type. You can override the default content type for the request by calling the **SetHeader** method prior to calling this method. Most servers require you to explicitly specify what type of data is being submitted by the client and will reject requests which do not correctly identify the content type.

It is common for servers to return additional information about a failed request in their response to the client. If you need to process this information, use the HTTP_POST_ERRORDATA option which causes the error message to be returned in the *lpvResult* buffer provided by the caller. If this option is used, your application should call **GetResultCode** to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This method will cause the current thread to block until the post completes, a timeout occurs or the operation is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be

periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Store the script output into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter. Since the output from the
// script is textual, the HTTP_POST_CONVERT option is used

nResult = lpClient->PostXml(lpszResource,
                            lpszXmlData,
                            &hgblBuffer,
                            &cbBuffer,
                            HTTP_POST_CONVERT);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableEvents](#), [GetData](#), [GetTransferStatus](#), [PostData](#), [PostJson](#), [PutData](#), [RegisterEvent](#)

CHttpClient::ProxyConnect Method

```
BOOL ProxyConnect(  
    UINT nProxyType,  
    LPCTSTR LpszProxyHost,  
    UINT nProxyPort,  
    LPCTSTR LpszProxyUser,  
    LPCTSTR LpszProxyPassword,  
    LPCTSTR LpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    DWORD dwVersion,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

```
BOOL ProxyConnect(  
    LPCTSTR LpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    DWORD dwVersion,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **ProxyConnect** method establishes a connection through a proxy server.

Parameters

nProxyType

An unsigned integer which specifies the type of proxy that the client is connecting to. The supported proxy server types are as follows:

Constant	Description
HTTP_PROXY_NONE	A direct connection will be established with the server. When this value is specified the proxy parameters are ignored.
HTTP_PROXY_STANDARD	A standard connection is established through the specified proxy server, and all resource requests will be specified using a complete URL. This proxy type should be used with standard connections.
HTTP_PROXY_SECURE	A secure connection is established through the specified proxy server. This proxy type should be used with secure connections and the HTTP_OPTION_SECURE option should also be set via the <i>dwOptions</i> parameter.
HTTP_PROXY_WINDOWS	The configuration options for the current system should be used. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. These settings are the same proxy server

lpszProxyHost

A pointer to a string which specifies the proxy server host name or IP address. This argument is ignored if the proxy type is set to HTTP_PROXY_NONE or HTTP_PROXY_WINDOWS and no proxy configuration has been specified for the local system.

nProxyPort

The port number that the proxy server is listening for connections on. A value of zero specifies that the default port number 80 should be used. Note that in most cases, a proxy server is not configured to use the default port. This argument is ignored if the proxy type is set to HTTP_PROXY_NONE or HTTP_PROXY_WINDOWS and no proxy configuration has been specified for the local system.

lpszProxyUser

A pointer to a string which specifies the user name that will be used to authenticate the client session to the proxy server. If the server does not require user authentication, then a NULL pointer may be passed in this argument.

lpszProxyPassword

A pointer to a string which specifies the password that will be used to authenticate the client session to the proxy server. If the server does not require user authentication, then a NULL pointer may be passed in this argument.

lpszRemoteHost

A pointer to a string which specifies the name of the server to connect to through the proxy server. This may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 80. For secure connections, the default port number is 443.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested. If the server does not support the keep-alive option, the client will automatically reconnect when each resource is requested.

	Although it will not provide any performance benefits, this allows the option to be used with all servers.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_ERRORDATA	This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
HTTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

dwVersion

The requested protocol version used when sending requests to the server. The high word should specify the major version, and the low word should specify the minor version number. The HTTPVERSION macro can be used to create version value.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur. If this argument is NULL, then the client session will be blocking and no network events will be sent to the client.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the *hEventWnd* argument is NULL, this argument should be specified as WM_NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The use of Windows event notification messages has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

To prevent this method from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **ProxyConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

If the HTTP_PROXY_WINDOWS proxy type is specified, then the proxy configuration for the local system is used. If no proxy server has been defined, then the proxy-related parameters will be ignored and the method will establish a connection directly to the server. The second form of the **ProxyConnect** method always uses the the system configuration to determine if the connection should be made through a proxy server, which is why there are no parameters such as the proxy type or proxy server name.

If you specify an event notification window, then the client session will be asynchronous. When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
HTTP_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
HTTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
HTTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of

	the data. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
HTTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
HTTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
HTTP_EVENT_PROGRESS	The client is in the process of sending or receiving data from the server. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.
HTTP_EVENT_REDIRECT	This event is generated when a the server indicates that the requested resource has been moved to a new location. The new resource location may be on the same server, or it may be located on another server.

To cancel asynchronous notification and return the client to a blocking mode, use the **DisableEvents** method.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **HttpAttachThread** function.

Specifying the HTTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Example

```
CHttpClient *pClient = new CHttpClient();

// Establish a connection using the default proxy server
// configuration for the local system

if (pClient->ProxyConnect(strHostName) == FALSE)
{
    pClient->ShowError();
    return;
}

// Retrieve the resource from the server and store it
// in the string buffer

nResult = pClient->GetData(strResource, strBuffer);

if (nResult == HTTP_ERROR)
    pClient->ShowError();

pClient->Disconnect();
delete pClient;
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [Disconnect](#), [IsConnected](#)

CHttpClient::PutData Method

```
INT PutData(  
    LPCTSTR lpszRemoteFile,  
    LPBYTE lpBuffer,  
    DWORD dwLength  
);
```

```
INT PutData(  
    LPCTSTR lpszRemoteFile,  
    LPCTSTR lpszBuffer  
);
```

The **PutData** method copies the contents of the specified buffer and stores it in a file on the server.

Parameters

lpszRemoteFile

A pointer to a string that specifies the file on the server that will be created. This string may specify a valid URL for the current server that the client is connected to.

lpBuffer

A pointer to a buffer which will contain the data to be transferred from the client and stored in a file on the server. In an alternate form of the method, this may be a null terminated string.

dwLength

The number of bytes to copy from the buffer.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **PutData** method is used to store the contents of the specified buffer in a file on the server. Not all servers permit files to be created using this method, and some may require that specific configuration changes be made to the server in order to support this functionality. Consult your server's technical reference documentation to see if it supports the PUT command, and if so, what must be done to enable it. It may be required that the client authenticate itself using the **Authenticate** method prior to uploading the data.

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a transfer while it is in progress, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttp10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableEvents](#), [GetData](#), [GetFile](#), [GetTransferStatus](#), [PostData](#), [PutFile](#), [RegisterEvent](#)

CHttpClient::PutFile Method

```
INT PutFile(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszRemoteFile  
);
```

The **PutFile** method transfers the specified file on the local system to the server.

Parameters

lpszLocalFile

A pointer to a string that specifies the file that will be transferred from the local system. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A pointer to a string that specifies the file on the server that will be created, overwritten or appended to. The file naming conventions must be that of the host operating system.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **PutFile** method is used to transfer a file from the local system to a server. Not all servers permit files to be uploaded and some may require that specific configuration changes be made to the server in order to support this functionality. Consult your server's technical reference documentation to see if it supports the PUT command, and if so, what must be done to enable it. It may be required that the client authenticate itself using the **Authenticate** method prior to uploading the file.

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableEvents](#), [GetData](#), [GetFile](#), [GetTransferStatus](#), [PostData](#), [PostFile](#), [PutData](#), [RegisterEvent](#)

CHttpClient::PutText Method

```
INT PutText(  
    LPCTSTR lpszRemoteFile,  
    LPCTSTR lpszBuffer  
);
```

The **PutText** method creates a text file on the server using the contents of a string buffer.

Parameters

lpszRemoteFile

A pointer to a string that specifies the text file on the server that will be created or overwritten. This string may specify a valid URL for the current server that the client is connected to.

lpszBuffer

A pointer to a string that contains the text that will be stored in the file.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **PutText** method is used to create a text file on the server from the contents of a string. If the specified file already exists on the server, its contents will be overwritten. Not all servers permit files to be created using this method, and some may require that specific configuration changes be made to the server in order to support this functionality. Consult your server's technical reference documentation to see if it supports the PUT command, and if so, what must be done to enable it. It may be required that the client authenticate itself using the **Authenticate** method prior to uploading the data.

If the Unicode version of this method is called, the string will be converted to ASCII and then uploaded to the server. If you wish to store the contents of the string as Unicode on the server, you must use the **PutData** method. This method should never be used to upload binary data.

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a file transfer while it is in progress, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableEvents](#), [GetText](#), [GetTransferStatus](#), [PutData](#), [PutFile](#), [RegisterEvent](#)

CHttpClient::Read Method

```
INT Read(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Read(  
    CString& strBuffer,  
    INT cbBuffer  
);
```

The **Read** method reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be copied. An alternate form of this method allows a **CString** variable to be passed and data read from the socket will be returned in that string.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When **Read** is called and the client is in non-blocking mode, it is possible that the method will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

See Also

[EnableEvents](#), [IsBlocking](#), [IsReadable](#), [IsWritable](#), [RegisterEvent](#), [Write](#)

CHttpClient::RegisterEvent Method

```
INT RegisterEvent(  
    UINT nEventId,  
    HTTPEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Constant	Description
HTTP_EVENT_CONNECT	The connection to the server has completed.
HTTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
HTTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
HTTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
HTTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
HTTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **HttpEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This method is typically used to register an event handler that is invoked while a resource is being retrieved or data is being submitted to the server. The HTTP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one HTTP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the *lpEventProc* parameter must be declared using the **_stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

See Also

[DisableEvents](#), [EnableEvents](#), [FreezeEvents](#), [GetTransferStatus HttpEventProc](#),

CHttpClient::SetBearerToken Method

```
INT SetBearerToken(  
    LPTSTR lpszBearerToken  
);
```

The **SetBearerToken** method sets the OAuth 2.0 bearer token used to authenticate the client session with a web service.

Parameters

lpszBearerToken

A pointer to a null terminated string buffer which contains the bearer token used to authorize client requests. If this parameter is NULL or a zero length string, the current bearer token will be cleared and no client authentication will be performed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

Using OAuth 2.0 requires you to understand the process of how to request the bearer (access) token. Obtaining a bearer token requires registering your application with the web service provider, getting a unique client ID associated with your application and then requesting the token using the appropriate scope for the service. Obtaining the initial token will typically involve interactive confirmation on the part of the user, requiring they grant permission to your application to access the service.

Your application should not store a bearer token for later use. They have a relatively short lifespan, typically about an hour, and are designed to be used with that session. You should specify offline access as part of the OAuth 2.0 scope if necessary and store the refresh token provided by the service. The refresh token has a much longer validity period and can be used to obtain a new bearer token when needed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Authenticate](#), [GetBearerToken](#), [SetHeader](#)

CHttpClient::SetCookie Method

```
BOOL SetCookie(  
    LPCTSTR lpszCookieName,  
    LPCTSTR lpszCookieValue  
);
```

The **SetCookie** method sends the specified cookie to the server when a resource is requested.

Parameters

lpszCookieName

Pointer to a string which specifies the name of the cookie that will be sent to the server when the next resource is requested.

lpszCookieValue

Pointer to a string which specifies the value of the cookie. To delete a cookie that has been previously set, this parameter should be NULL or point to an empty string.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetCookie** method submits the cookie name and value to the server when a resource is requested or data is posted to a script. For more information about cookies and how they are used, refer to the **GetCookie** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

See Also

[GetCookie](#), [GetFirstCookie](#), [GetNextCookie](#), [SetHeader](#)

CHttpClient::SetEncodingType Method

```
INT SetEncodingType(  
    INT nEncodingType  
);
```

The **SetEncodingType** method specifies the type of encoding to be applied to the content of a HTTP request.

Parameters

nEncodingType

Specifies the content encoding type; currently-supported values are:

Constant	Description
HTTP_ENCODING_NONE	No encoding will be applied to the content of a request, and no Content-Type header will be generated.
HTTP_ENCODING_URL	URL encoding will be applied to the content of a request, and a Content-Type header will be generated with the value "application/x-www-form-urlencoded"
HTTP_ENCODING_XML	URL encoding will be applied to the content of a request, EXCEPT that spaces will not be replaced by '+'. This encoding type is intended for use with XML parsers that do not recognize '+' as a space. A Content-Type header will be generated with the value "application/x-www-form-urlencoded".

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SetEncodingType** method explicitly sets the type of encoding used when optional parameter data is submitted with a request for a resource. By default, data is URL encoded and the content type will be designated as application/x-www-form-urlencoded.

If an application specifies HTTP_ENCODING_NONE, then the parameter data is not encoded and there is no content type header created by default. The client application can create its own Content-Type header field by calling the **SetHeader** method if necessary.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Command](#), [GetEncodingType](#), [GetData](#), [PostData](#), [SetHeader](#)

CHttpClient::SetFormProperties Method

```
INT SetFormProperties(  
    LPHTTPFORMPROPERTIES LpFormProp  
);
```

The **SetFormProperties** function updates the properties of the current form.

Parameters

LpFormProp

Points to a HTTPFORMPROPERTIES structure which specifies the new properties for the current form.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateForm](#), [GetFormProperties](#), [SubmitForm](#), [HTTPFORMPROPERTIES](#)

CHttpClient::SetHeader Method

```
BOOL SetHeader(  
    LPCTSTR lpszHeader,  
    LPCTSTR lpszValue  
);
```

The **SetHeader** method sets the value of the specified request header field.

Parameters

lpszHeader

Points to a string which specifies the request header field name.

lpszValue

Points to a string which specifies the value of the request header field.

Return Value

If the method succeeds, the return value is non-zero. If the client handle is invalid, the method returns a value of zero. To get extended error information, call **GetLastError**.

Remarks

The **SetHeader** method defines a header field and value that is submitted to the server when a resource is requested. If the header field has already been defined, it will be replaced by the new value. There are a number of header fields which are automatically created by the library, and others that are conditionally created depending on whether or not certain options are specified. The following header fields are automatically created by the library:

Header Field	Description
Accept	This header specifies the types of resources that are acceptable to the client. By default, all resource data types are accepted by the client.
Authorization	This header is automatically created if the client uses the Authenticate method to authenticate a client session.
Connection	This header determines if the connection is maintained after a resource has been requested, or if the connection should be immediately closed. The value of this header depends on whether the HTTP_OPTION_KEEPALIVE option has been specified.
Content-Length	This header defines the length of the data that is being posted to the server or the size of a file being uploaded to the server.
Content-Type	This header defines the content type for data posted to the server. This header is created if URL encoding is specified. The specific type of encoding used can be set by calling the SetEncodingType method.
Host	This header specifies the name of the server that the client has connected to. This is automatically generated when any resource is requested.
Pragma	This header is used to control caching performed by the server. If

	the option HTTP_OPTION_NOCACHE has been specified, this header will automatically be defined with the value "no-cache".
Proxy-Authorization	This header is automatically created if a proxy connection has been established and a username and password is required to authenticate the client session.

Request headers are generated by methods that send resource requests. In some cases, header values are supplied by the requesting method only if the application has not previously defined the header. For others, the requesting method overrides what the application may have defined.

If you use this method to set the Authorization header to a custom value for this client session, you must not call the **Authenticate** method. The **Authenticate** method will always override any custom Authorization header value and replace it with the credentials token generated from the username and password provided.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Authenticate](#), [GetFirstHeader](#), [GetHeader](#), [GetNextHeader](#)

CHttpClient::SetLastError Method

```
VOID SetLastError(  
    DWORD dwErrorCode  
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or HTTP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttp10.lib

See Also

[GetErrorString](#), [GetLastError](#), [ShowError](#)

CHttpClient::SetOption Method

```
INT SetOption(  
    DWORD dwOption,  
    BOOL bEnabled  
);
```

The **SetOption** method enables or disables the specified option.

Parameters

hClient

Handle to the client session.

dwOption

An unsigned integer which specifies one of the following options:

Constant	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested. If the server does not support the keep-alive option, the client will automatically reconnect when each resource is requested. Although it will not provide any performance benefits, this allows the option to be used with all servers.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_NOUSERAGENT	This option specifies the client should not include a User-Agent header with any requests made during the session. The user agent is a string which is used to identify the client application to the server.
HTTP_OPTION_ERRORDATA	This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.
HTTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not

	limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

bEnabled

An integer value which determines if the option should be enabled or disabled. A non-zero value specifies that the option should be enabled, while a zero value specifies that the option should be disabled.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SetOption** method can only enable or disable the options listed above. Other options are only available when the connection is initially established. For example, you cannot use this function to enable security options after the connection has been made.

If you use HTTP_OPTION_FREETHREAD to permit any thread to reference a client handle allocated in another thread, your application is responsible for ensuring it does not attempt to submit requests using the same handle on different threads at the same time. If two different threads attempt to perform an operation using the same handle, there is no guarantee as to which thread will complete the operation first.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

See Also

[Connect](#), [GetOption](#), [ProxyConnect](#)

CHttpClient::SetPriority Method

```
INT SetPriority(  
    INT nPriority  
);
```

The **SetPriority** method specifies the priority for file transfers.

Parameters

nPriority

An integer value which specifies the new priority for file transfers. It may be one of the following values:

Constant	Description
HTTP_PRIORITY_NORMAL	The default priority which balances resource utilization and transfer speed. It is recommended that most applications use this priority.
HTTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the transfer. It is typically used with worker threads running in the background when the amount of time required perform the transfer is not critical.
HTTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the transfer and meters the bandwidth allocated for the transfer. This priority will increase the average amount of time required to complete a file transfer.
HTTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the transfer, allocating more memory for internal buffering. It can be used when it is important to transfer the file quickly, and there are no other threads currently performing file transfers at the time.
HTTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization while attempting to transfer the file as quickly as possible. If the file transfer is being performed in the main UI thread, this priority can cause the application to appear to become non-responsive. No events will be generated during the transfer.

Return Value

If the method succeeds, the return value is the previous file transfer priority. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SetPriority** method can be used to control the processor usage, memory and network bandwidth allocated for file transfers. The default priority balances resource utilization and transfer speed while ensuring that a single-threaded application remains responsive to the user. Lower priorities reduce the overall resource utilization at the expense of transfer speed. For example, if

you create a worker thread to download a file in the background and want to ensure that it has a minimal impact on the process, the `HTTP_PRIORITY_BACKGROUND` value can be used.

Higher priority values increase the memory allocated for the transfers and increases processor utilization for the transfer. The `HTTP_PRIORITY_CRITICAL` priority maximizes transfer speed at the expense of system resources. It is not recommended that you increase the file transfer priority unless you understand the implications of doing so and have thoroughly tested your application. If the file transfer is being performed in the main UI thread, increasing the priority may interfere with the normal processing of Windows messages and cause the application to appear to become non-responsive. It is also important to note that when the priority is set to `HTTP_PRIORITY_CRITICAL`, normal progress events will not be generated during the transfer.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshttpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetPriority](#)

CHttpClient::SetTimeout Method

```
INT SetTimeout(  
    UINT nTimeout  
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

See Also

[Connect](#), [GetTimeout](#), [IsReadable](#), [IsWritable](#), [Read](#), [Write](#)

CHttpClient::ShowError Method

```
INT ShowError(  
    LPCTSTR lpszAppTitle,  
    UINT uType,  
    DWORD dwErrorCode  
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A pointer to a string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetErrorString](#), [GetLastError](#), [SetLastError](#)

CHttpClient::SubmitForm Method

```
INT SubmitForm(  
    LPBYTE lpResult,  
    LPDWORD lpcbResult,  
    DWORD dwOptions  
);
```

```
INT SubmitForm(  
    HGLOBAL* lpResult,  
    LPDWORD lpcbResult,  
    DWORD dwOptions  
);
```

```
INT SubmitForm(  
    CString& strResult,  
    DWORD dwOptions  
);
```

The **SubmitForm** method submits the contents of the current form to a script on the server and returns the result in a buffer provided by the caller.

Parameters

lpResult

A pointer to a byte buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvBuffer* parameter. If the *lpvResult* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of data that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_SUBMIT_DEFAULT	The default post mode. The contents of the buffer are encoded and sent as standard form data. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
HTTP_SUBMIT_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SubmitForm** method is used to submit form data to a script that executes on the server and then copy the output from that script into a local buffer. The method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the resulting output of the script. In this case, the *lpvResult* parameter will point to the buffer that was allocated by the client and the value that the *lpcbResult* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvResult* parameter point to a global memory handle which will contain the data when the function returns. In this case, the value that the *lpcbResult* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

This method will cause the current thread to block until the post completes, a timeout occurs or the operation is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** function.

To determine the current status of a transaction while it is in progress, use the **GetTransferStatus** function.

Example

```
CString strResult;
INT nResult = 0;

pClient->CreateForm(_T("/login.php"), HTTP_METHOD_POST, HTTP_FORM_ENCODED);
pClient->AddField(_T("UserName"), lpszUserName);
pClient->AddField(_T("Password"), lpszPassword);

nResult = pClient->SubmitForm(strResult);
pClient->DestroyForm();
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AddField](#), [AddFile](#), [ClearForm](#), [CreateForm](#), [DeleteField](#), [RegisterEvent](#)

CHttpClient::UploadFile Method

```
BOOL UploadFile(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszFileURL,  
    UINT nTimeout  
    DWORD dwOptions  
    LPHTTPTRANSFERSTATUS lpStatus  
    HTTPEVENTPROC lpEventProc  
    DWORD_PTR dwParam  
);
```

The **UploadFile** method uploads the specified file from the local system to the server.

Parameters

lpszLocalFile

A pointer to a string that specifies the file on the local system that will be uploaded to the server. The file pathing and name conventions must be that of the local host.

lpszFileURL

A pointer to a string that specifies the complete URL of the file that will be created or overwritten on the server. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed using the system Control Panel.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.

HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.

lpStatus

A pointer to an HTTPTRANSFERSTATUS structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **UploadFile** method provides a convenient way for an application to upload a file in a single function call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session and then upload the file to the server. The URL must be complete, and specify either a standard or secure HTTP scheme:

```
[http|https]://[username : password] @] remotehost [:remoteport] / [path / ...] [filename]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. If a path is specified as part of the URL, then function will attempt to change the current working directory. Note that the path in an HTTP URL is relative to the home directory of the user account and is not an absolute path starting at the root directory on the server. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "http" scheme is used and the HTTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, the "https" scheme must be specified.

Not all web servers permit files to be uploaded and some may require that specific configuration changes be made to the server in order to support this functionality. Consult your server's technical reference documentation to see if it supports the PUT command, and if so, what must be done to enable it. It may be required that the URL specify a username and password to upload a file.

The *lpStatus* parameter can be used by the application to determine the final status of the transfer, including the total number of bytes copied, the amount of time elapsed and other information related to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The *lpEventProc* parameter specifies a pointer to a function which will be periodically called during the file transfer process. This can be used to check the status of the transfer by calling **GetTransferStatus** and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The *dwParam* parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the *this* pointer as the value, and then cast it back to an object pointer inside the callback function. If no event handler is required, then a NULL pointer can be specified as the value for *lpEventProc* and the *dwParam* parameter will be ignored.

The **UploadFile** method is designed to provide a simpler interface for uploading a file. However, complex connections such as those using a proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **Connect** and then use **PutFile** to upload the file. If the server does not support the PUT command, you may be able to upload files using the **PostFile** method. Refer to that method for more information.

Example

```
CHttpClient httpClient;  
CString strLocalFile = _T("c:\\temp\\database.mdb");  
CString strFileURL =  
_T("http://update:secret@www.example.com/updates/database.mdb");  
  
if (!httpClient.UploadFile(strLocalFile, strFileURL))  
{  
    httpClient.ShowError();  
    return;  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DownloadFile](#), [GetTransferStatus](#), [HttpEventProc](#), [PostFile](#), [PutFile](#), [HTTPTRANSFERSTATUS](#)

CHttpClient::ValidateUrl Method

```
BOOL CHttpClient::ValidateUrl(  
    LPCTSTR lpszURL  
);
```

The **ValidateUrl** method determines if a string represents a valid HTTP URL.

Parameters

lpszURL

A pointer to a string that specifies the URL to validate.

Return Value

If the specified URL is valid and the host name can be resolved to an IP address, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **ValidateUrl** method will check the value of a string to ensure that it represents a complete, valid URL using either a standard or secure HTTP scheme. This method will not establish a connection with the server to verify that it exists, it will only attempt to resolve the host name to an IP address. If the remote host is specified as an IP address, this method will check to make sure that the address is formatted correctly. Note that if you wish to specify an IPv6 address, you must enclose the address in brackets.

To establish a connection with a server using a URL, use the **ConnectUrl** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ConnectUrl](#), [DownloadFile](#), [UploadFile](#)

CHttpClient::VerifyFile Method

```
BOOL VerifyFile(  
    LPCTSTR lpzLocalFile,  
    LPCTSTR lpzRemoteFile  
);
```

The **VerifyFile** method attempts to verify that the size of a file on the local system is the same as the specified file on the server.

Parameters

lpzLocalFile

A pointer to a string that specifies the name of file on the local system.

lpzRemoteFile

A pointer to a string that specifies the name of the file on the server.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **VerifyFile** method will attempt to verify that the contents of the local and remote files are identical by comparing the size of the files. This method is provided for compatibility with the File Transfer Protocol API, and should not be considered a reliable method for comparing files. Web servers may not consistently return file size information for dynamically created content such as HTML pages which use server-side includes.

It is not recommended that you use this method with text files because of the different end-of-line conventions used by different operating systems. For example, a text file on a Windows system uses a carriage-return and linefeed pair to indicate the end of a line of text. However, on a UNIX system, a single linefeed is used to indicate the end of a line. This can cause the **VerifyFile** method to indicate the files are not identical, even though the only difference is in the end-of-line characters that are used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DeleteFile](#), [GetFile](#), [GetFileSize](#), [PutFile](#)

CHttpClient::Write Method

```
INT Write(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Write(  
    LPCTSTR lpszBuffer  
    INT cbBuffer  
);
```

The **Write** method sends the specified number of bytes to the server.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server. In an alternate form of the method, the pointer is to a string.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero, unless a pointer to a null-terminated string is passed as the buffer argument. In that case, if the value is -1, all of the characters in the string, up to but not including the terminating null character, will be sent to the server.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the next asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableEvents](#), [IsBlocking](#), [IsReadable](#), [IsWritable](#), [Read](#), [RegisterEvent](#)

Hypertext Transfer Protocol Data Structures

- HTTPFORMPROPERTIES
- HTTPTRANSFERSTATUS
- HTTPTRANSFERSTATUSEX
- SECURITYCREDENTIALS
- SECURITYINFO
- SYSTEMTIME

HTTPFORMPROPERTIES Structure

This structure is used by the **GetFormProperties** and **SetFormProperties** methods to return and modify the properties of the specified form.

```
typedef struct _HTTPFORMPROPERTIES
{
    UINT    nFormMethod;
    UINT    nFormType;
    LPCTSTR lpszFormAction;
    DWORD   dwReserved1;
    DWORD   dwReserved2;
} HTTPFORMPROPERTIES, *LPHTTPFORMPROPERTIES;
```

Members

nFormMethod

An unsigned integer value which specifies how the form data will be submitted to the server. It may be one of the following values:

Constant	Description
HTTP_METHOD_GET	The form data should be submitted using the GET command. This method should be used when the amount of form data is relatively small. If the total amount of form data exceeds 2048 bytes, it is recommended that the POST method be used instead.
HTTP_METHOD_POST	The form data should be submitted using the POST command. This is the preferred method of submitting larger amounts of form data. If the total amount of form data exceeds 2048 bytes, it is recommended that the POST method be used.

nFormType

An unsigned integer value which specifies the type of form and how the data will be encoded when it is submitted to the server. It may be one of the following values:

Constant	Description
HTTP_FORM_ENCODED	The form data should be submitted as URL encoded values. This is typically used when the GET method is used to submit the data to the server.
HTTP_FORM_MULTIPART	The form data should be submitted as multipart form data. This is typically used when the POST method is used to submit a file to the server. Note that the script must understand how to process multipart form data if this form type is specified.

lpszFormAction

A pointer to a string which specifies the name of the resource that the form data will be submitted to. Typically this is the name of a script that is executed on the server.

dwReserved1

A reserved structure member.

dwReserved2

A reserved structure member.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateForm](#), [GetFormProperties](#), [SetFormProperties](#), [SubmitForm](#)

HTTPTRANSFERSTATUS Structure

This structure is used by the **GetTransferStatus** method to return information about a data transfer in progress.

```
typedef struct _HTTPTRANSFERSTATUS
{
    DWORD    dwBytesTotal;
    DWORD    dwBytesCopied;
    DWORD    dwBytesPerSecond;
    DWORD    dwTimeElapsed;
    DWORD    dwTimeEstimated;
} HTTPTRANSFERSTATUS, *LPHTTPTRANSFERSTATUS;
```

Members

dwBytesTotal

The total number of bytes that will be transferred. If the data is being downloaded from the server to the local host, this is the size of the requested resource. If the data is being uploaded from the local host to the server, it is the size of the local file or memory buffer. If the size of the resource cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the data transfer is completed. This is based on the average number of bytes transferred per second.

Remarks

If the option `HTTP_OPTION_HIRES_TIMER` has been specified when connecting to the server, the values returned in the *dwTimeElapsed* and *dwTimeEstimated* members will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small amounts of data over a fast network connection.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

See Also

[EnableEvents](#), [GetTransferStatus](#), [RegisterEvent](#), [HTTPTRANSFERSTATUSEX](#)

HTTPTRANSFERSTATUSEX Structure

This structure is used by the **GetTransferStatus** method to return information about a data transfer in progress. This structure is designed for use with extended functions that support files larger than 4GB.

```
typedef struct _HTTPTRANSFERSTATUSEX
{
    ULARGE_INTEGER uiBytesTotal;
    ULARGE_INTEGER uiBytesCopied;
    DWORD          dwBytesPerSecond;
    DWORD          dwTimeElapsed;
    DWORD          dwTimeEstimated;
} HTTPTRANSFERSTATUSEX, *LPHTTPTRANSFERSTATUSEX;
```

Members

uiBytesTotal

The total number of bytes that will be transferred. If the data is being downloaded from the server to the local host, this is the size of the requested resource. If the data is being uploaded from the local host to the server, it is the size of the local file or memory buffer. If the size of the resource cannot be determined, this value will be zero.

uiBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the data transfer is completed. This is based on the average number of bytes transferred per second.

Remarks

If the option `HTTP_OPTION_HIRES_TIMER` has been specified when connecting to the server, the values returned in the *dwTimeElapsed* and *dwTimeEstimated* members will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small amounts of data over a fast network connection.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

See Also

[EnableEvents](#), [GetTransferStatus](#), [RegisterEvent](#), [HTTPTRANSFERSTATUS](#)

SECURITYCREDENTIALS Structure

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on

	what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that

will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A pointer to a null terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A pointer to a null terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A pointer to a null terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A pointer to a null terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
------------	-------------

CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
----	---

MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
----	--

ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.
------	--

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

A pointer to a null terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used for SSH connections

and should always be NULL when establishing a secure connection using SSL or TLS.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* member should be set to CREDENTIAL_STORE_FILENAME and the *lpzCertStore* member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS#12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the *lpzPassword* member or the library will be unable to access the certificate.

Note that the *lpzUserName* and *lpzPassword* members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

The TLS 1.1 and TLS 1.2 protocols are only supported on Windows 7, Windows Server 2008 R2 and later versions of the platform. If these options are specified and the application is running on Windows XP or Windows Vista, the protocol version will be downgraded to TLS 1.0 for backwards compatibility with those platforms.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include ctools10.h.

Unicode: Implemented as Unicode and ANSI versions.

SECURITYINFO Structure

This structure contains information about a secure connection that has been established.

```
typedef struct _SECURITYINFO
{
    DWORD        dwSize;
    DWORD        dwProtocol;
    DWORD        dwCipher;
    DWORD        dwCipherStrength;
    DWORD        dwHash;
    DWORD        dwHashStrength;
    DWORD        dwKeyExchange;
    DWORD        dwCertStatus;
    SYSTEMTIME   stCertIssued;
    SYSTEMTIME   stCertExpires;
    LPCTSTR      lpszCertIssuer;
    LPCTSTR      lpszCertSubject;
    LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to **sizeof(SECURITYINFO)** prior to passing the address of this structure to the function. Note that if this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the protocol that was selected to establish the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The

	<p>correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.</p>
SECURITY_PROTOCOL_TLS10	<p>The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS11	<p>The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS12	<p>The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.</p>

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_CIPHER_RC2	The RC2 block cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
SECURITY_CIPHER_DES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher, using 56-bit

	keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher (also known as the Rijndael cipher) is a fixed block size cipher which use a key size of 128, 192 or 256 bits. This cipher is supported on Windows XP SP3 and later versions of the operating system.
SECURITY_CIPHER_SKIPJACK	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
SECURITY_CIPHER_BLOWFISH	The Blowfish block cipher was selected. This is a variable key length cipher up to 448 bits, using a 64-bit block size.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected. One of the following values will be returned:

Constant	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected. One of the following values will be returned:

--	--

Constant	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography. This key exchange algorithm is only supported on Windows XP SP3 and later versions of the operating system.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server. This member only has meaning for connections using the SSL or TLS protocols. One of the following values will be returned:

Constant	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertIssuer

A pointer to a string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertSubject

A pointer to a string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszFingerprint

A pointer to a string which contains a sequence of hexadecimal values that uniquely identify the server. This member is only used when a connection has been established using the Secure Shell (SSH) protocol.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Unicode: Implemented as Unicode and ANSI versions.

SYSTEMTIME Structure

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include windows.h.

Hypertext Transfer Protocol Server Class

Implements a server that enables the application to send and receive files using the Hypertext Transfer Protocol.

Reference

- [Data Members](#)
- [Class Methods](#)
- [Event Handlers](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

File Name	CSHTSV10.DLL
Version	10.0.1468.2518
LibID	39B2930E-60DC-42D6-B388-6D6CD768DBE7
Import Library	CSHTSV10.LIB
Dependencies	None
Standards	RFC 1945, RFC 2616, RFC 3875

Overview

This library provides an interface for implementing an embedded, lightweight server that can be used to provide access to documents and other resources using the standard Hypertext Transfer Protocol. The server can accept connections from any standard web browser, third-party applications or programs developed using the SocketTools HTTP client API.

The application specifies an initial server configuration and then responds to events that are generated when the client sends a request to the server. An application may implement only minimal handlers for most events, in which case the default actions are performed for most standard HTTP commands. However, an application may also use the event mechanism to filter specific commands or to extend the protocol by providing custom implementations of existing commands or add entirely new commands.

The server includes support for CGI scripting, virtual hosting, client authentication and the creation of virtual directories and files. The server also supports secure connections using TLS. Secure connections require a valid security certificate to be installed on the system.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical

updates available for your version of the operating system.

This class provides an implementation of a multithreaded server which should only be used with languages that support the creation of multithreaded applications. It is important that you do not attempt to link against static libraries which were not built with support for threading.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

CHttpServer Public Data Members

Member Variables	Description
m_nCacheTime	The number of seconds until the client should consider a cached response to be stale
m_nExecTime	The maximum number of seconds that the server will permit an external command to execute
m_nIdleTime	The maximum number of seconds a client can be idle before the server terminates the session
m_nLogFormat	The format used by the server to log client activity
m_nLogLevel	The level of detail included in the server log file
m_nMaxClients	The maximum number of active client sessions accepted by the server
m_nMaxClientsPerAddress	The maximum number of clients per IP address accepted by the server
m_nMaxPostSize	The maximum amount of data a client may submit to the server
m_nMaxRequests	The maximum number of requests a client may make per connection
m_dwOptions	Options specified when creating an instance of the server
m_dwStackSize	The initial size of the stack allocated for threads created by the server

CHttpServer::m_nExecTime

```
UINT m_nExecTime;
```

The maximum number of seconds that the server will permit an external command to execute.

Remarks

The **m_nExecTime** data member is a public variable that specifies the maximum number of seconds that an external CGI program is permitted to run on the server. Programs are registered using the **RegisterProgram** method and are executed when the client sends a request for a resource that is associated with the program. If this value is zero, the default timeout period of 5 seconds will be used. The minimum execution time is 1 second and the maximum time limit is 30 seconds. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

[CHttpServer](#), [RegisterProgram](#)

CHttpServer::m_nIdleTime

`UINT m_nIdleTime;`

The maximum number of seconds a client can be idle before the server terminates the session.

Remarks

The **m_nIdleTime** data member is a public variable that specifies the maximum number of seconds that a client session may be idle before the server closes the control connection to the client. A value of zero specifies the default value of 60 seconds. If the value is non-zero, the minimum value is 10 seconds and the maximum value is 300 seconds (5 minutes). This value is used to initialize the default idle timeout period for each client session. The server determines if a client is idle based on the time the last command was issued and whether or not a file transfer is in progress. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

[CHttpServer](#), [RegisterProgram](#), [SetClientIdleTime](#)

CHttpServer::m_nLogFormat

UINT m_nLogFormat;

The format used when updating the server log file.

Remarks

The **m_nLogFormat** data member is a public variable that specifies the format of the log file that is created or updated by the server. It may be one of the following values:

Constant	Description
HTTP_LOGFILE_NONE (0)	This value specifies that the server should not create or update a log file.
HTTP_LOGFILE_COMMON (1)	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about file transfers.
HTTP_LOGFILE_COMBINED (2)	This value specifies that the server should use the combined log file format. This format is similar to the common format, however it includes the client referrer and user agent. This is the format that most Apache web servers use by default.
HTTP_LOGFILE_EXTENDED (3)	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session.

By default, logging is not enabled for the server. Changing the value of this data member does not have an effect on an active instance of the server. To change the format, level of detail or default log file name, use the **SetLogFile** method.

See Also

[CHttpServer](#), [GetLogFile](#), [SetLogFile](#)

CHttpServer::m_nLogLevel

`UINT m_nLogLevel;`

The level of detail included in the server log file.

Remarks

The **m_nLogLevel** data member is a public variable that specifies the level of detail that should be generated in the log file. The minimum value is 1 and the maximum value is 10. If the **m_nLogFormat** data member specifies a valid log file format and this value is zero, a default level of detail will be selected based on the format.

The common log file format generally contains less information by default, only logging the data transfers between the client and server. The W3C extended log file format defaults to a higher level of detail that includes additional information about the client session. The higher the level of detail, the larger the log file will be.

By default, logging is not enabled for the server. Changing the value of this data member does not have an effect on an active instance of the server. To change the format, level of detail or default log file name, use the **SetLogFile** method.

See Also

[CHttpServer](#), [GetLogFile](#), [SetLogFile](#)

CHttpServer::m_nMaxClients

`UINT m_nMaxClients;`

The maximum number of clients that are permitted to connect to the server.

Remarks

The **m_nMaxClients** data member is a public variable that specifies the maximum number of clients that are permitted to establish a connection with the server. After this limit is reached, the server will reject additional connections until the number of active clients drops below this threshold. A value of zero specifies that there is no fixed limit on the active number of client connections. Changing the value of this data member does not have an effect on an active instance of the server. To change the maximum number of clients on an active server, use the **Throttle** method.

The actual number of client connections that can be accepted depends on the amount of memory available to the server process. Sockets are allocated from the non-paged memory pool, so the actual number of sockets that can be created system-wide depends on the amount of physical memory that is installed. If the server will be accessible over the Internet, it is recommended that you limit the maximum number of client connections to a reasonable value.

See Also

[CHttpServer](#), [Throttle](#)

CHttpRequest::m_nMaxClientsPerAddress

`UINT m_nMaxClientsPerAddress;`

The maximum number of clients that are permitted to connect to the server from a single IP address.

Remarks

The **m_nMaxClientsPerAddress** data member is a public variable that specifies the maximum number of clients that are permitted to establish a connection with the server from a single IP address. After this limit is reached, the server will reject additional connections until the number of active clients drops below this threshold. A value of zero specifies that there is no limit on the active number of client connections per IP address. Changing the value of this data member does not have an effect on an active instance of the server. To change the maximum number of clients on an active server, use the **Throttle** method.

It is not recommended that you set the maximum clients per address below a value of 4. Lower values can negatively impact the performance of some clients, and may result in unexpected errors.

See Also

[CHttpServer](#), [Throttle](#)

CHttpRequest::m_dwOptions

DWORD m_dwOptions;

The default options used when starting an instance of the server.

Remarks

The **m_dwOptions** data member is a public variable that specifies the default options that should be used when starting an instance of the server. This variable can be modified directly or by calling the **SetOptions** method. For a list of available server options, see [Server Option Constants](#). Changing the value of this data member does not have an effect on an active instance of the server.

See Also

[CHttpServer](#), [GetOptions](#), [SetOptions](#)

CHttpServer Methods

Class	Description
CHttpServer	Constructor which initializes the current instance of the class
~CHttpServer	Destructor which releases resources allocated by the class
Method	Description
AddVirtualHost	Add a new virtual host to the server virtual host table
AddVirtualHostAlias	Add an alternate host name for an existing virtual host
AddVirtualPath	Add a new virtual path for the specified host
AddVirtualUser	Add a new virtual user for the specified host
AsyncNotify	Enable or disable asynchronous notification of changes in server status
AttachHandle	Attach the specified server handle to this instance of the class
AuthenticateClient	Authenticate the client and assign access rights for the session
CheckVirtualPath	Determine if the client has permission to access the specified virtual path
DeleteAllClientHeaders	Delete all of the request or response headers for the specified client session
DeleteClientHeader	Delete a request or response header for the specified client session
DeleteVirtualHost	Delete a virtual host associated with the specified server
DeleteVirtualPath	Remove a virtual path from the specified host
DeleteVirtualUser	Delete a virtual user from the specified host
DetachHandle	Detach the server handle from the current instance of this class
DisableCommand	Disable a specific server command
DisableTrace	Disable the logging of network function calls
DisconnectClient	Disconnect the specific client session, closing the control channel and aborting any file transfer
EnableClientAccess	Enable or disable access rights for the specified client session
EnableCommand	Enable a specific server command
EnableTrace	Enable logging of network function calls to a file
EnumClients	Returns a list of active client connections established with the server
GetActiveClient	Return the client ID for the active client session associated with the current thread
GetAddress	Return the IP address for the server
GetAllClientHeaders	Return all client request or response headers in a string buffer
GetClientAccess	Return the access rights that have been granted to the client session
GetClientAddress	Return the IP address of the specified client session
GetClientCredentials	Return the credentials for the specified client session

GetClientDirectory	Return the root document directory for a client session
GetClientHeader	Return the value of a request or response header for the specified client session
GetClientIdleTime	Return the idle timeout period for the specified client
GetClientLocalPath	Return the full local path for the specified virtual path
GetClientServer	Return the handle to the server that created the specified client session
GetClientThreadId	Returns the thread ID associated with the specified client session
GetClientUserName	Return the user name associated with the specified client session
GetClientVariable	Return the value of a CGI environment variable for the specified client
GetClientVirtualHost	Return the name of the virtual host the client used to establish the connection
GetClientVirtualHostId	Return the virtual host ID associated with the specified client session
GetClientVirtualPath	Return the virtual path for a local file on the server
GetCommandFile	Return the full path to the local file name or directory specified by the client
GetCommandLine	Return the complete command line issued by the client
GetCommandName	Return the name of the command that was issued by the client
GetCommandQuery	Return the query parameters included with the command
GetCommandResource	Return the path for the resource requested by the client
GetCommandResult	Return the result code and a description of the last command processed by the server
GetCommandUrl	Return the complete URL of the resource requested by the client
GetDirectory	Return the full path to the root directory assigned to the specified server
GetHandle	Return the server handle associated with the class instance
GetIdentity	Return the identity and version information for the specified server
GetLastError	Return information about the last server error that occurred
GetLogFile	Return the current log file format and full path for the file
GetMemoryUsage	Return the amount of memory allocated for the server and all client sessions
GetName	Return the host name assigned to the server or specified client session
GetOptions	Return the options specified for this instance of the server
GetPriority	Return the current priority assigned to the specified server
GetProgramExitCode	Return the exit code of the last program executed by the client
GetProgramName	Return the name of the CGI program executed by the client
GetProgramOutput	Return a copy of the standard output from the last program executed by the client
GetProgramText	Return a copy of the standard output from the last program in a string buffer
GetStackSize	Return the initial size of the stack allocated for threads created by the server
GetTransferInfo	Return information about the current file transfer

GetUuid	Return the UUID assigned to the specified server
GetVirtualHostId	Return the virtual host ID associated with the specified hostname
GetVirtualHostName	Return the hostname associated with the specified virtual host ID
IsActive	Determine if the server has been started
IsClientAuthenticated	Determine if the specified client session has been authenticated
IsCommandEnabled	Determine if the specified command is currently enabled or disabled
IsInitialized	Determine if the class has been successfully initialized
IsListening	Determine if the server is listening for client connections
PreProcessEvent	Filter server events before being processed by the default event handler
ReceiveRequest	Receive the request that was sent by the client to the server
RedirectRequest	Redirect the request from the client to another URL
RegisterHandler	Register a CGI program for use and associate it with a file name extension
RegisterProgram	Register a CGI program for use and associate it with a virtual path on the server
RenameServerLogFile	Rename or delete the current log file being updated by the server
RequireAuthentication	Send a response to the client indicating that authentication is required
Restart	Restart the server, terminating all active client sessions
Resume	Resume accepting client connections on the specified server
SendErrorResponse	Send a customized error response to the specified client
SendResponse	Send a response from the server to the specified client
SendResponseData	Send additional data to the client in response to a command
SetCertificate	Set the name of the certificate to be used with secure connections.
SetClientAccess	Change the access rights associated with the specified client session
SetClientHeader	Create or change the value of a request or response header for the client session
SetClientIdleTime	Change the idle timeout period for the specified client session
SetClientVariable	Create or change the value of a CGI environment variable for the specified client
SetCommandFile	Change the name of the local file or directory that is the target of the current command
SetDirectory	Specify the local directory that will be used as the server root directory
SetLastError	Set the last error code for the specified server session
SetLogFile	Change the current log format, level of detail and file name
SetIdentity	Change the identity and version information for the specified server
SetName	Change the hostname assigned to the server or specified client session
SetOptions	Change the options specified for this instance of the server
SetPriority	Change the priority assigned to the specified server

SetStackSize	Change the initial size of the stack allocated for threads created by the server
SetUuid	Assign a UUID to be associated with this instance of the server
Start	Start the server and begin accepting client connections
Stop	Stop the server and terminate all active client connections
Suspend	Suspend accepting client connections on the specified server
Throttle	Limit the number of active client connections, connections per address and connection rate

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CHttpServer::~CHttpServer

`~CHttpServer();`

The **CHttpServer** destructor releases resources allocated by the current instance of the **CHttpServer** object. It also uninitialized the library if there are no other concurrent uses of the class.

Remarks

When a **CHttpServer** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshtsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CHttpServer](#)

CHttpServer::AddVirtualHost Method

```
UINT HttpAddVirtualHost(  
    LPCTSTR lpszHostName,  
    UINT nHostPort,  
    LPCTSTR lpszDirectory  
);
```

Add a new virtual host to the server virtual host table.

Parameters

lpszHostName

A pointer to a string which specifies the hostname that will be added to the virtual host table. This parameter must specify a valid hostname and cannot be a NULL pointer or a zero-length string.

nHostPort

An integer value which specifies the port number for the virtual host. This value must be zero or the same value as the original port number that the server was configured to use.

lpszDirectory

A pointer to a NULL terminated string which specifies the root document directory for the virtual host. If this parameter is NULL or a zero-length string, the virtual host will use the same root directory that was specified when the server was started. This parameter may contain environment variables enclosed in % symbols.

Return Value

If the method is successful, it will return a non-zero integer value that identifies the virtual host. If the method fails, it will return INVALID_VIRTUAL_HOST and the last error code will be updated to indicate the cause of the failure.

Remarks

Virtual hosting is a method for sharing multiple domain names on a single instance of a server. The client provides the server with the hostname that it has used to establish the connection, and that name is compared against a table of virtual hosts configured for the server. If the hostname matches a virtual host, the client will use the root directory and any virtual paths that have been assigned to that host.

When the server is first started, a default virtual host with an ID of zero is automatically created and is identified as VIRTUAL_HOST_DEFAULT. This virtual host uses the same hostname, port number and root directory that the server instance was created with. The application should treat all other host IDs as opaque values and never make assumptions about how they are allocated.

The *nHostPort* parameter should always be specified with a value of zero, or the same port number that the server was configured to use. Port-based virtual hosting is currently not supported and this parameter is included for future use.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AddVirtualHostAlias](#), [AddVirtualPath](#), [DeleteVirtualHost](#), [GetVirtualHostId](#), [GetVirtualHostName](#)

CHttpServer::AddVirtualHostAlias Function

```
BOOL AddVirtualHostAlias(  
    UINT nHostId,  
    LPCTSTR lpszHostAlias  
);  
BOOL AddVirtualHostAlias(  
    LPCTSTR lpszHostAlias  
);
```

Add an alternate host name for an existing virtual host.

Parameters

nHostId

An integer value which identifies the virtual host. If this parameter is omitted, the alias will be assigned to the default server.

lpszHostAlias

A pointer to a string which specifies the alias for the virtual host. The alias must be a valid domain name that uniquely identifies the host. This parameter cannot be a NULL pointer or specify an empty string.

Return Value

If the function succeeds, the return value is non-zero. If the virtual host ID does not specify a valid host, the function will return zero. If the function fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **AddVirtualHostAlias** method adds an alias for an existing virtual host. This enables a client to establish a connection using a number of different domain names which all reference the same virtual host. When the server responds to the client, it will identify itself with the primary domain name assigned to the virtual host rather than the alias provided by the client.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AddVirtualHost](#), [AddVirtualPath](#)

CHttpServer::AddVirtualPath Function

```
BOOL AddVirtualPath(  
    UINT nHostId,  
    LPCTSTR lpszVirtualPath,  
    LPCTSTR lpszLocalPath,  
    DWORD dwFileAccess  
);
```

Add a new virtual path for the specified host.

Parameters

nHostId

An integer value which identifies the virtual host. If this parameter is omitted, the default virtual host will be used.

lpszVirtualPath

A pointer to a string which specifies the virtual path that will be created. This parameter cannot be a NULL pointer or an empty string. The maximum length of the virtual path is 1024 characters.

lpszLocalPath

A pointer to a string which specifies the local directory or file name that the virtual path will be mapped to. This path must exist and can be no longer than MAX_PATH characters. This parameter cannot be a NULL pointer or an empty string.

dwFileAccess

An integer value which specifies the access clients will be given to the virtual path. For a list of file access permissions, see [User and File Access Constants](#).

Return Value

If the method succeeds, the return value is non-zero. If the virtual host ID does not specify a valid host, the method will return zero. If the method fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **AddVirtualPath** method maps a virtual path name to a directory or file name on the local system. Virtual paths are assigned to specific hosts and if multiple virtual hosts are created for the server, each can have its own virtual paths which map to different files. To create a virtual path for the default server, the *nHostId* parameter should be specified as VIRTUAL_HOST_DEFAULT.

It is recommended that the *lpszLocalPath* parameter always specify the full path to the local file or directory. If the path is relative, it will be considered relative to the current working directory for the process and expanded to its full path name. The local path can include environment variables surrounded by % symbols. For example, if the value %ProgramData% is included in the path, it will be expanded to the full path for the common application data folder. The local path cannot specify a Windows system folder or the root directory of a mounted drive volume.

The local file or directory does not need to be located in the document root directory for the server or virtual host. It can specify any valid local path that the server process has the appropriate permissions to access. You should exercise caution when creating virtual paths to files or directories outside of the server root directory. If the *lpszLocalPath* parameter specifies a directory, clients will have access to that directory and all subdirectories using its virtual path.

If you wish to password protect the virtual file or directory, include the `HTTP_ACCESS_PROTECTED` flag in the file permissions. The default command handlers will recognize this flag and require that the client authenticate itself to grant access to the resource. If the server application implements a custom command handler, it is responsible for checking for the presence of this flag and perform the appropriate checks to ensure that the client session has been authenticated.

If the server was started in restricted mode, the client will be unable to access documents outside of the server root directory and its subdirectories. This restriction also applies to virtual paths that reference documents or other resources outside of the root directory. To allow a client to access a document outside of the server root directory, the **SetClientAccess** method should be used to grant the client `HTTP_ACCESS_READ` permission.

The **GetClientVirtualPath** method will return the virtual path that is associated with a local file or directory. The **GetClientLocalPath** method will return the full path to a local file or directory that is mapped to a virtual path.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshtsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AddVirtualHost](#), [CheckVirtualPath](#), [GetClientLocalPath](#), [GetClientVirtualPath](#), [DeleteVirtualPath](#)

CHttpServer::AddVirtualUser Method

```
BOOL AddVirtualUser(  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    DWORD dwUserAccess,  
    LPCTSTR lpszDirectory  
);
```

```
BOOL AddVirtualUser(  
    UINT nHostId,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    DWORD dwUserAccess,  
    LPCTSTR lpszDirectory  
);
```

Add a new virtual user for the specified host.

Parameters

nHostId

An integer value which identifies the virtual host. If this parameter is omitted, the default virtual host will be used.

lpszUserName

A pointer to a string which specifies the user name. The maximum length of a username is 63 characters and it is recommended that names be limited to alphanumeric characters. Whitespace, control characters and certain symbols such as path delimiters and wildcard characters are not permitted. If an invalid character is included in the name, the method will fail with an error indicating the username is invalid. This parameter cannot be NULL and the name must be at least three characters in length. Usernames are not case sensitive.

lpszPassword

A pointer to a string which specifies the user password. The maximum length of a password is 63 characters and is limited to printable characters. Whitespace and control characters are not permitted. If an invalid character is included in the password, the method will fail with an error indicating the password is invalid. This parameter cannot be NULL and must be at least one character in length. Passwords are case sensitive.

dwUserAccess

An integer value which specifies the access clients will be given when authenticated as this user. For a list of user access permissions, see [User and File Access Constants](#). If this parameter is omitted, default access rights will be assigned based on the server configuration.

lpszDirectory

A pointer to a string which specifies the local directory that is considered to be the virtual user's home directory. This path must exist and can be no longer than MAX_PATH characters. The maximum length of the local path is 260 characters. The directory cannot be located in a Windows system folder or the root directory of a mounted disk volume. If this parameter is omitted, the server root directory will be assigned as the user home directory.

Return Value

If the method succeeds, the return value is non-zero. If the the virtual host ID does not specify a valid host, or the username or password contain invalid characters, the method will return zero. If

the method fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **AddVirtualUser** method adds a virtual user that is associated with the specified virtual host. If a client attempts to access a protected document and provides credentials, the server will attempt to automatically authenticate the session by searching for virtual user with the same username and password. If a match is found, then the client session is assigned the same access permissions as the virtual user.

If the server is started with the HTTP_SERVER_MULTUSER option, then documents in the virtual user's home directory can be accessed by specifying their username using a specially formatted request URL. For example, if a virtual user named "Thomas" is created, the documents in that user's home directory could be accessed as `http://servername/~thomas/document.html`

All files and subdirectories in the user's home directory are considered to be read-only. A client cannot create files in a user's home directory, even if they are authenticated as that user. In addition, CGI programs and scripts cannot be executed from a user's home directory.

If you wish to modify the information for a user, it is not necessary to delete the username first. If this method is called with a username that already exists, that record is replaced with the values passed to this method.

The virtual users created by this method exist only as long as the server is active. If you wish to maintain a persistent database of users and passwords, you are responsible for its implementation based on the requirements of your specific application. For example, a simple implementation would be to store the user information in a local XML or INI file and then read that configuration file after the server has started, calling this method for each user that is listed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshtsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AddVirtualHost](#), [DeleteVirtualUser](#)

CHttpServer::AsyncNotify Method

```
BOOL AsyncNotify(  
    HWND hWnd,  
    UINT uMsg  
);
```

Enable or disable asynchronous notification of changes in server status.

Parameters

hWnd

A handle to the window whose window procedure will receive the notification message.

uMsg

The user-defined message that will be sent to the notification window.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **AsyncNotify** method is used by an application to enable or disable asynchronous notifications. The message window is typically the main UI window and these notifications are used signal to the application that it should update the user interface. If the *hWnd* parameter is not NULL, it must specify a valid window handle and the user-defined message must have a value of **WM_USER** or higher. The application cannot specify a notification message that is reserved by the operating system. The pseudo-handle **HWND_BROADCAST** cannot be specified as the notification window. If the *hWnd* parameter is NULL, notifications for the specified server will be disabled.

When asynchronous notifications are enabled for a server, the server will post the user-defined message to the window whenever there is a change in status or after a client has connected or disconnected from the server. The *wParam* message parameter will contain the notification message and the *lParam* message parameter will contain the handle to the server or the client ID. The following notification messages are defined:

Constant	Description
HTTP_NOTIFY_STARTUP	This notification is sent when the server has started and is preparing to accept client connections. This notification is only sent once, and only if asynchronous notifications are enabled immediately after the Start method is called. This message will not be sent once the server has begun accepting client connections or when notification messages are disabled and then subsequently re-enabled at a later time. The <i>lParam</i> message parameter will specify the handle to the server.
HTTP_NOTIFY_LISTEN	This notification is sent when the server is listening for client connections. This notification message may be sent to the application multiple times over the lifetime of the server. If the server was suspended, this notification will be sent after the application calls the

	Resume method to resume accepting client connections. The <i>lParam</i> message parameter will specify the handle to the server.
HTTP_NOTIFY_SUSPEND	This notification is sent when the server suspends accepting new connections because the application has called the Suspend method. This notification message may be sent to the application multiple times over the lifetime of the server. The <i>lParam</i> message parameter will specify the handle to the server.
HTTP_NOTIFY_RESTART	This notification is sent when the server is restarted using the Restart method. Note that the server socket handle provided by the <i>lParam</i> message parameter will specify the new socket handle of the restarted server instance, not the original socket handle. The <i>lParam</i> message parameter will specify the handle to the server.
HTTP_NOTIFY_CONNECT	This notification is sent when the server accepts a client connection and the thread that manages the client session has begun processing network events for that client. This message notification will not be sent if the client connection is rejected by the server. The <i>lParam</i> message parameter will specify the unique ID of the client that connected to the server.
HTTP_NOTIFY_DISCONNECT	This notification is sent when the client disconnects from the server and the client socket has been closed. This notification message may not occur for each client session that is forced to terminate as the result of the server being stopped using the Stop method. The <i>lParam</i> message parameter will specify the unique ID of the client that disconnected from the server.
HTTP_NOTIFY_SHUTDOWN	This notification is sent when the server thread is in the process of terminating. At the time the application processes this notification message, the server handle in <i>lParam</i> will reference the defunct server and cannot be used with other server methods. The <i>lParam</i> message parameter will specify the handle to the server.

If asynchronous notifications are enabled, you should never use those notifications as a replacement for an event handler. When an event occurs, the callback function that handles the event is invoked in the context of the thread that manages the client session. The application should exchange data with the client within that event handler and not in response to a notification message. These notification messages should only be used to update the application UI in response to changes in the status of the server.

The HTTP_NOTIFY_CONNECT and HTTP_NOTIFY_DISCONNECT notifications are different from the other server notifications because the *lParam* message parameter does not specify the server handle, but rather the unique client ID associated with the session that connected to or disconnected from the server. Use the **GetClientServer** method to obtain a handle to the server that created the client session. Note that at the time the application processes the HTTP_NOTIFY_DISCONNECT notification message, the client session will have already terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cstools10.h.

Import Library: cshtsv10.lib

See Also

[GetClientServer](#), [Start](#)

CHttpServer::AttachHandle Method

```
VOID AttachHandle(  
    HSERVER hServer  
);
```

The **AttachHandle** method attaches the specified server handle to the current instance of the class.

Parameters

hServer

The handle to the server that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a server handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that server.

If a server handle already has been created for the class, that handle will be released when the new handle is attached to the class object. This will cause the server to stop and all client sessions will be terminated immediately. If you want to prevent the previous server from being stopped, you must call the **DetachHandle** method prior to attaching a new handle to the class instance.

Note that the *hServer* parameter is presumed to be a valid server handle and no checks are performed to ensure that the handle references an active server. Specifying an invalid server handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

See Also

[DetachHandle](#), [GetHandle](#)

CHttpServer::AuthenticateClient Method

```
BOOL AuthenticateClient(  
    UINT nClientId,  
    DWORD dwUserAccess  
);
```

Authenticate the client and assign access rights for the session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies one or more user access rights. For a list of user access rights that can be granted to the client, see [User and File Access Constants](#).

Return Value

If the the client session could be authenticated, the return value is non-zero. If the client ID does not specify a valid client session, or the client has already been authenticated, this method will return zero.

Remarks

The **HttpAuthenticateClient** method is used to authenticate a specific client session, typically in response to an **OnAuthenticate** event that indicates a client has provided authentication credentials as part of the request for a document or other resource.

To enable the server to automatically authenticate a client session, use the **AddVirtualUser** method to add one or more virtual users. The server will search the list of virtual users for a match to the credentials provided by the client and will set the appropriate permissions for the session without requiring a event handler to manually authenticate the session using this method.

If the server was started with the HTTP_SERVER_LOCALUSER option and the client session is not authenticated using this method, the server will attempt to authenticate the client session using the local Windows user database. Although this option can be convenient because it does not require the implementation of an event handler for the **OnAuthenticate** event, it can be used by clients to attempt to discover valid usernames and passwords for the local system. It is recommended that you use the **AddVirtualUser** method to create virtual users rather than using the local user database.

It is recommended that most applications specify HTTP_ACCESS_DEFAULT as the *dwUserAccess* value for a client session, since this allows the server automatically grant the appropriate access based on the server configuration options.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AddVirtualUser](#), [GetClientCredentials](#), [GetClientDirectory](#), [OnAuthenticate](#)

CHttpServer::CheckVirtualPath Method

```
BOOL CheckVirtualPath(  
    UINT nClientId,  
    LPCTSTR lpszVirtualPath,  
    DWORD dwFileAccess  
);
```

Determine if the client has permission to access the specified virtual path.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszVirtualPath

A pointer to a string which specifies the virtual path that should be checked. This path must be absolute and cannot be a NULL pointer or an empty string. The maximum length of the virtual path is 1024 characters.

dwFileAccess

An unsigned integer value which specifies the access permissions that should be checked. For a list of file access permissions, see [User and File Access Constants](#).

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client, the method will return zero. If the method fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **CheckVirtualPath** method is used to determine if the client has permission to access the virtual file or directory, based on the value of the *dwFileAccess* parameter. For example, if the *dwFileAccess* parameter has the value HTTP_ACCESS_WRITE, this method will check if the client has write permission for the file or directory. The method will return a non-zero value if the client does have the requested permission, or zero if it does not.

Applications that implement their own custom handlers for standard HTTP commands should use this method to ensure that the client has the appropriate permissions to access the requested resource. Failure to check the access permissions for the client can result in the client being able to access restricted documents and other resources. It is recommended that most applications use the default command handlers.

To obtain the path to the local file or directory that the virtual path is mapped to, use the **GetClientLocalPath** method.

Example

```
CString strPathName;  
  
// Get the current request URL path  
INT cchPathName = pServer->GetCommandResource(nClientId, strPathName);  
  
if (cchPathName == 0)  
{  
    pServer->SendErrorResponse(nClientId, 500);  
    return;  
}
```

```
}  
  
// Check if the client has write access to that resource  
BOOL bAllowed = pServer->CheckVirtualPath(  
    nClientId,  
    strPathName,  
    HTTP_ACCESS_WRITE);  
  
if (!bAllowed)  
{  
    pServer->SendErrorResponse(nClientId, 403);  
    return;  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AddVirtualPath](#), [DeleteVirtualPath](#), [GetClientLocalPath](#), [GetClientVirtualPath](#)

CHttpServer::CHttpServer Method

`CHttpServer();`

The **CHttpServer** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **HttpServerInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** method because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshtsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[~CHttpServer](#), [IsInitialized](#)

CHttpRequest::DeleteAllClientHeaders Method

```
BOOL DeleteAllClientHeaders(  
    UINT nClientId,  
    UINT nHeaderType  
);
```

Delete all of the request or response headers for the specified client session.

Parameters

nClientId

Delete all of the request or response headers for the specified client session.

nHeaderType

Specifies the type of headers to delete. It may be one of the following values:

Constant	Description
HTTP_HEADERS_REQUEST	Delete all of the request headers that were provided by the client. Request header values provide additional information to the server about the type of request being made.
HTTP_HEADERS_RESPONSE	Delete all of the response headers that were created by the server in response to a request made by the client. Response header values provide additional information to the client about the type of information that is being returned by the server.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **DeleteAllClientHeaders** method is used to delete all of the request or response headers that were set as the result of the client issuing a request for a document or other resource. If this method is used to delete all of the response headers, the server will automatically generate a standard set of response headers when it returns the requested information to the client.

It is not necessary to call this method inside an **OnDisconnect** event handler to delete the header values that were set during the client session. This is done automatically when the client disconnects from the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[DeleteClientHeader](#), [SetClientHeader](#)

CHttpServer::DeleteClientHeader Function

```
BOOL DeleteClientHeader(  
    UINT nClientId,  
    UINT nHeaderType,  
    LPCTSTR lpszHeaderName  
);
```

Delete a request or response header for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nHeaderType

Specifies the type of header to delete. It may be one of the following values:

Constant	Description
HTTP_HEADERS_REQUEST	Delete a request header that was provided by the client. Request header values provide additional information to the server about the type of request being made.
HTTP_HEADERS_RESPONSE	Delete a response header that was created by the server. Response header values provide additional information to the client about the type of information that is being returned by the server.

lpszHeaderName

A pointer to a string that specifies the name of the header that should be deleted. Header names are not case-sensitive and should not include the colon which acts as a delimiter that separates the header name from its value. This parameter cannot be a NULL pointer or an empty string.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero. If the method fails, the **GetLastError** method will return more information about the last error that has occurred.

Remarks

The **DeleteClientHeader** method will delete a request or response header for the specified client session. There are a number of required response headers that are always sent to a client and deleting the header using this method will cause the server to automatically generate a new default header value. You should not delete response header values unless you are certain of the impact that it would have on the normal operation of the client.

It is not necessary for you to delete a header value to change the value of an existing header. The **SetClientHeader** method will replace an existing header value with a new value, or create a new header if the header name does not already exist.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DeleteAllClientHeaders](#), [SetClientHeader](#)

CHttpServer::DeleteVirtualHost Method

```
BOOL DeleteVirtualHost(  
    UINT nHostId  
);
```

Delete a virtual host associated with the specified server.

Parameters

nHostId

An integer value which identifies the virtual host.

Return Value

If the method succeeds, the return value is non-zero. If the server handle is invalid or the virtual host ID does not specify a valid host, the method will return zero. If the method fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **DeleteVirtualHost** method removes a virtual host that was created by a previous call to the **AddVirtualHost** method. All virtual paths and users associated with the specified host are no longer valid. It is not necessary to call this method to delete any of the virtual hosts prior to stopping the server. Part of the normal shutdown process is releasing the resources allocated for each virtual host that was added to the server.

This method cannot be used to delete the virtual host with an ID of zero, which is the default virtual host that is allocated when the server is started.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

See Also

[AddVirtualHost](#), [AddVirtualPath](#), [AddVirtualUser](#) [DeleteVirtualPath](#), [DeleteVirtualUser](#)

CHttpServer::DeleteVirtualPath Method

```
BOOL DeleteVirtualPath(  
    UINT nHostId,  
    LPCTSTR lpszVirtualPath  
);
```

Remove a virtual path from the specified host.

Parameters

nHostId

An integer value which identifies the virtual host.

lpszVirtualPath

A pointer to a string which specifies the virtual path that will be removed. This path must be absolute and cannot be a NULL pointer or an empty string.

Return Value

If the method succeeds, the return value is non-zero. If the virtual host ID does not specify a valid host, the method will return zero. If the method fails, the last error code will be updated to indicate the cause of the failure.

Remarks

This method removes a virtual path that was created by a previous call to the **AddVirtualPath** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AddVirtualHost](#), [AddVirtualPath](#), [GetClientLocalPath](#), [GetClientVirtualPath](#)

CHttpServer::DeleteVirtualUser Method

```
BOOL DeleteVirtualUser(  
    UINT nHostId,  
    LPCTSTR lpszUserName  
);
```

Remove a virtual user from the specified host.

Parameters

nHostId

An integer value which identifies the virtual host.

lpszUserName

A pointer to a string which specifies the user that will be removed. This parameter cannot be a NULL pointer or an empty string.

Return Value

If the method succeeds, the return value is non-zero. If the the virtual host ID does not specify a valid host, or the username does not exist, the method will return zero. If the method fails, the last error code will be updated to indicate the cause of the failure.

Remarks

This method removes a virtual user that was created by a previous call to the **AddVirtualUser** method. This method will not match partial usernames and wildcard characters cannot be used to delete multiple users. Usernames are not case sensitive.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AddVirtualHost](#), [AddVirtualUser](#)

CHttpServer::DetachHandle Method

```
HSERVER DetachHandle();
```

The **DetachHandle** method detaches the server handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the server handle associated with the current instance of the class object. If there is no active server, the value `INVALID_SERVER` will be returned.

Remarks

This method is used to detach a server handle created by the class for use with the SocketTools API. Once the server handle is detached from the class, no other class member functions may be called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshtsv10.lib`

See Also

[AttachHandle](#), [GetHandle](#)

CHttpServer::DisableCommand Method

```
BOOL EnableCommand(  
    LPCTSTR LpszCommand  
);
```

Disable a specific server command.

Parameters

lpszCommand

A pointer to a NULL terminated string that specifies the name of the command to be disabled. The command name is not case-sensitive, but the value must otherwise match the exact name. Partial matches are not recognized by this method. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the command is not recognized, the method will return zero. If the method fails, the **GetLastError** method will return more information about the last error that has occurred.

Remarks

The **DisableCommand** method is used to disable access to a specific command on the server, typically for security purposes. For example, the PUT command can be disabled, preventing any client from attempting to upload files directly to the server. The **IsCommandEnabled** method can be used to determine if a command is enabled or not.

The command name provided to this method must match the commands defined in RFC 2616 or related protocol standards. Refer to [Hypertext Transfer Protocol Commands](#) for a complete list of server commands.

Some commands cannot be disabled because they are required to perform essential server functions. For example, the GET and HEAD commands cannot be disabled. If you attempt to disable a required command, this method will return zero and the last error code will be set to ST_ERROR_COMMAND_REQUIRED. Because this method affects all clients connected to the server, it should not be used to limit access to certain commands for specific clients. Instead, use an event handler to filter the commands.

The OPTIONS and TRACE commands are disabled by default for all server instances and must be explicitly enabled using the **EnableCommand** method if you wish permit clients to use them. It is not recommended that you enable these commands if your server is going to be publicly accessible over the Internet. If the server started with the option HTTP_SERVER_READONLY, commands that can be used to create or modify files on the server will be disabled by default.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AuthenticateClient](#), [EnableCommand](#), [IsCommandEnabled](#)

CHttpServer::DisableTrace Method

```
BOOL DisableTrace();
```

Disable the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[EnableTrace](#)

CHttpServer::DisconnectClient Method

```
BOOL DisconnectClient(  
    UINT nClientId  
);
```

Close the control connection for the specified client and release the resources allocated for the session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **DisconnectClient** method will close the control channel, disconnecting the client from the server and terminating the client session thread. Resources that we allocated for the client, such as memory and open handles, will be released back to the operating system. If the client was in the process of transferring a file, the transfer will be aborted. This performs the same operation as if the client sent the QUIT command to the server.

This method sends an internal control message that notifies the server that this session should be terminated. When the session thread is signaled that it should terminate, it will abort any active file transfers and begin to release the resources allocated for that session. To ensure that the client session terminates gracefully, there may be a brief period of time where the session thread is still active after this method has returned.

After this method returns, the application should never use the same client ID with another method. Client IDs are unique to the session over the lifetime of the server, and are not reused.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[Restart](#), [Stop](#)

CHttpServer::EnableClientAccess Method

```
BOOL EnableClientAccess(  
    UINT nClientId,  
    DWORD dwUserAccess,  
    BOOL bEnable  
);
```

Enable or disable access rights for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies an access right to enable or disable. For a list of user access rights that can be granted to the client, see [User and File Access Constants](#).

bEnable

An integer value which specifies if permission should be granted or revoked for the specified access right. If this value is non-zero, permission is granted to the client to perform the action specified by the *dwUserAccess* parameter. If this value is zero, that permission is revoked.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **EnableClientAccess** method is used to enable or disable access to specific functionality by the client. The method can only change a single access right and cannot be used to enable or disable multiple access rights in a single method call. To change multiple user access rights for the client, use the **SetClientAccess** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[AuthenticateClient](#), [GetClientAccess](#), [SetClientAccess](#)

CHttpServer::EnableCommand Method

```
BOOL EnableCommand(  
    LPCTSTR lpszCommand  
);
```

Enable a specific server command.

Parameters

lpszCommand

A pointer to a NULL terminated string that specifies the name of the command to be enabled or disabled. The command name is not case-sensitive, but the value must otherwise match the exact name. Partial matches are not recognized by this method. This parameter cannot be NULL.

bEnable

An integer value which specifies if the command should be enabled or disabled. If the value is non-zero, the command is enabled. If the value is zero, the command will be disabled.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero. If the method fails, the **GetLastError** method will return more information about the last error that has occurred.

Remarks

The **EnableCommand** method is used to enable access to a specific command on the server. The **IsCommandEnabled** method can be used to determine if a command is enabled or not. The command name provided to this method must match the commands defined in RFC 2616 or related protocol standards. Refer to [Hypertext Transfer Protocol Commands](#) for a complete list of server commands.

The OPTIONS and TRACE commands are disabled by default for all server instances and must be explicitly enabled using the **EnableCommand** method if you wish permit clients to use them. It is not recommended that you enable these commands if your server is going to be publicly accessible over the Internet. If the server started with the option HTTP_SERVER_READONLY, commands that can be used to create or modify files on the server will be disabled by default.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AuthenticateClient](#), [DisableCommand](#), [IsCommandEnabled](#)

CHttpRequest::EnableTrace Method

```
BOOL EnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

Enable the logging of network function calls to a file.

Parameters

lpszTraceFile

A pointer to a string that specifies the name of the log file. If this parameter is NULL or points to an empty string, a log file is created in the temporary directory for the current user.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_DEFAULT (0)	All function calls are written to the trace file. This is the default value.
TRACE_ERROR (1)	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING (2)	Only those function calls which fail, or return values which indicate a warning, are recorded in the trace file.
TRACE_HEXDUMP (4)	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

When trace logging is enabled, the log file is opened, appended to and closed for each socket function call. Using the same file name, you can do the same in your application to add additional information to the file if needed. This can provide an application-level context for the entries made by the library. Make sure that the file is closed after the data has been written. If a file name is not specified by the caller, a file named **cstrace.log** will be created in the temporary directory for the current user.

The TRACE_HEXDUMP option can produce very large files, since all data that is being sent and received by the application is logged. To reduce the size of the file, you can enable and disable logging around limited sections of code that you wish to analyze.

To redistribute an application that includes this debug logging functionality, the **cstrcv10.dll** library must be included as part of the installation package. This library provides the trace logging features, and if it is not available the **EnableTrace** method will fail. Note that this is a standard Windows DLL and does not need to be registered, it only needs to be redistributed with your application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableTrace](#)

CHttpRequest::EnumClients Method

```
INT EnumClients(  
    UINT * lpClients,  
    INT nMaxClients  
);
```

Return a list of active client connections established with the server.

Parameters

lpClients

Pointer to an array of unsigned integers which will contain client IDs that uniquely identifies each client when the method returns. If this parameter is NULL, then the method will return the number of active client connections established with the server.

nMaxClients

Maximum number of client IDs to be returned in the *lpClients* array. If the *lpClients* parameter is NULL, this parameter should have a value of zero.

Return Value

If the method succeeds, the return value is the number of active client connections to the server. If the method fails, the return value is HTTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the *nMaxClients* parameter is less than the number of active client connections, the method will fail and the last error code will be set to the error ST_ERROR_BUFFER_TOO_SMALL. To dynamically determine the number of active connections, call the method with the *lpClients* parameter with a value of NULL, and the *nMaxClients* parameter with a value of zero.

Example

```
// Populate a listbox with all of the users connected to the server  
pListBox->ResetContent();  
  
INT nClients = pHttpRequest->EnumClients();  
if (nClients > 0)  
{  
    UINT *pIdList = new UINT[nClients];  
  
    nClients = pHttpRequest->EnumClients(pIdList, nClients);  
    if (nClients == HTTP_ERROR)  
    {  
        // Unable to obtain list of connected clients  
        return;  
    }  
  
    for (INT nIndex = 0; nIndex < nClients; nIndex++)  
    {  
        CString strUserName;  
  
        if (pHttpRequest->GetClientUserName(pIdList[nIndex], strUserName))  
            pListBox->AddString(strUserName);  
    }  
  
    // Free the memory allocated for the client IDs
```

```
    delete pIdList;  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[GetClientAddress](#), [GetClientDirectory](#), [GetClientUserName](#)

CHttpRequest::GetActiveClient Method

```
UINT GetActiveClient();
```

Return the client ID for the active client session associated with the current thread.

Parameters

None.

Return Value

If the method succeeds, the return value is the unique ID associated with the client session for the current thread. If there is no client session active on the current thread, the return value is zero.

Remarks

The **GetActiveClient** method is used to obtain the client ID associated with the current thread. This means this method will only return a client ID if it is called within an event handler or a method called by an event handler. If this method is called by a function that is not executing within the context of an event handler it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[EnumClients](#)

CHttpRequest::GetAddress Method

```
INT GetAddress(  
    UINT nClientId,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

```
INT GetAddress(  
    UINT nClientId,  
    CString& strAddress  
);
```

Return the IP address of the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszAddress

A pointer to a string buffer that will contain the server IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the client ID is invalid, or the buffer is not large enough to store the complete address, the method will return a value of zero.

Remarks

This method will return the IP address assigned to the specified server as a printable string. If the *nClientId* parameter has a value of zero, this method will return the IP address assigned to the local system. If the HTTP_SERVER_NATROUTER option was specified when the server was started, this method will return the external IP address assigned to the system. If the *nClientId* parameter specifies a valid client session, this method will return the IP address that the client used to establish the connection with the server. To determine the IP address assigned to the client, use the **GetClientAddress** method.

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientAddress](#), [GetName](#)

CHttpRequest::GetClientAccess Method

```
BOOL GetClientAccess(  
    UINT nClientId,  
    DWORD& dwUserAccess  
);
```

Return the access rights that have been granted to the client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies one or more access rights for the client session. For a list of user access rights that can be granted to the client, see [User and File Access Constants](#).

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero. This method can only be used with authenticated clients. If the client session has not been authenticated, the return value will be zero.

Remarks

The **GetClientAccess** method is used to obtain all of the access rights that are currently granted to an authenticated client session. The **EnableClientAccess** method can be used to enable or disable specific permissions, and the **SetClientAccess** method can change multiple access rights at once.

Example

```
DWORD dwUserAccess = 0;  
  
// Check if the client has execute permission  
if (pHttpRequest->GetClientAccess(nClientId, dwUserAccess))  
{  
    if (dwUserAccess & HTTP_ACCESS_EXECUTE)  
    {  
        std::cout << "Client can execute programs and scripts\n";  
        return;  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[AuthenticateClient](#), [EnableClientAccess](#), [SetClientAccess](#)

CHttpServer::GetClientAddress Method

```
INT GetClientAddress(  
    UINT nClientId,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

```
INT GetClientAddress(  
    UINT nClientId,  
    CString& strAddress  
);
```

Return the IP address of the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszAddress

A pointer to a string buffer that will contain the client IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID is invalid, or the buffer is not large enough to store the complete address, the method will return a value of zero.

Remarks

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetAddress](#)

CHttpServer::GetClientCredentials Method

```
BOOL GetClientCredentials(  
    UINT nClientId,  
    LPHTTPCLIENTCREDENTIALS lpCredentials  
);  
  
BOOL GetClientCredentials(  
    UINT nClientId,  
    CString& strUserName,  
    CString& strPassword  
);
```

Return the user credentials for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpCredentials

A pointer to an **HTTPCLIENTCREDENTIALS** structure that will contain information about the user when the method returns. This parameter cannot be NULL.

strUserName

A string that will contain the user name when the method returns. This version of the method is only available for MFC and ATL based projects that define the **CString** object.

strPassword

A string that will contain the user password when the method returns. This version of the method is only available for MFC and ATL based projects that define the **CString** object.

Return Value

If the user credentials for the client session are available, the return value is non-zero. If the client ID does not specify a valid client session, or the client has not requested authentication, this method will return zero.

Remarks

The **GetClientCredentials** method is used to obtain the username and password that was provided by the client when it requested authentication. Typically this method is used in an event handler to validate the credentials provided by the client. If the credentials are considered valid, the event handler would then call the **AuthenticateClient** method to specify that the session has been authenticated.

If the default event handler is used, the **OnAuthenticate** method will be invoked with the user credentials passed to the handler as arguments.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

CHttpServer::GetClientDirectory Method

```
INT GetClientDirectory(  
    UINT nClientId,  
    LPTSTR lpszDirectory,  
    INT nMaxLength  
);  
  
INT GetClientDirectory(  
    UINT nClientId,  
    CString& strDirectory  
);
```

Returns the root document directory for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszDirectory

A pointer to a string buffer that will contain the root document directory for the specified client session, terminated with a null character. This buffer should be at least MAX_PATH characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be larger than zero or the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID does not specify a valid client session, the method will return zero.

Remarks

This method returns the full path to the root document directory for the specified client session. If no virtual hosts have been configured, then this value will be the same as the root directory assigned to the server when it was started. If the server has been configured with multiple virtual hosts, this function will return the path to the root directory associated with the hostname provided by the client.

To convert a full path to the virtual path for a specific client session, use the **GetClientVirtualPath** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientVirtualPath](#)

CHttpServer::GetClientHeader Method

```
INT GetClientHeader(  
    UINT nClientId,  
    UINT nHeaderType,  
    LPCTSTR lpszHeaderName,  
    LPTSTR lpszHeaderValue,  
    INT nMaxLength,  
);
```

Return the value of a request or response header for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nHeaderType

Specifies the type of header. It may be one of the following values:

Constant	Description
HTTP_HEADERS_REQUEST	Return the value of a request header that was provided by the client. Request header values provide additional information to the server about the type of request being made.
HTTP_HEADERS_RESPONSE	Return the value of a response header that was created by the server. Response header values provide additional information to the client about the type of information that is being returned by the server.

lpszHeaderName

A pointer to a string that specifies the name of the header field. Header names are not case-sensitive and should not include the colon which acts as a delimiter that separates the header name from its value. This parameter cannot be a NULL pointer or an empty string.

lpszHeaderValue

A pointer to a buffer that will contain the header value, terminated with a null character. To determine the length of the header value, this parameter can be NULL and the *nMaxLength* parameter should be specified with a value of zero.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the header value buffer, including the terminating null character. If the *lpszHeaderValue* parameter is NULL, this value must be zero.

Return Value

If the method succeeds, the return value is the length of the header value, not including the terminating null character. If the client ID does not specify a valid client session, or there is no header that matches the given name, the method will return zero. If the *lpszHeaderValue* parameter is not NULL and the buffer is not large enough to store the complete header value, the method will return zero and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL. If the method fails, the **GetLastError** method will return more information about the last error that has occurred.

Remarks

The **GetClientHeader** method will return the value of a request or response header for the specified client session. If the *lpszHeaderName* value matches an existing header field, its value will be copied to the string buffer provided by the caller.

Refer to [Hypertext Transfer Protocol Headers](#) for a list of common request and response headers that are used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DeleteClientHeader](#), [GetAllClientHeaders](#), [SetClientHeader](#)

CHttpRequest::GetClientIdleTime Method

```
UINT GetClientIdleTime(  
    UINT nClientId,  
    UINT * lpnElapsed  
);
```

Return the idle timeout period for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpnElapsed

An optional pointer to an unsigned integer value that will contain the number of seconds the client session has been idle. This parameter may be NULL or omitted if this information is not required.

Return Value

If the method succeeds, the return value is client idle timeout period in seconds. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **GetClientIdleTime** method will return the number of seconds that the client may remain idle before being automatically disconnected by the server. The idle time of a client session is based on the last time a command was issued to the server or when a data transfer completed. The server will never disconnect a client that is in the process of sending or receiving data, regardless of the idle timeout period.

The default idle timeout period for a client session is 60 seconds, however the server can be configured to use a different value. The minimum timeout period for a client is 10 seconds, the maximum is 300 seconds (5 minutes). An application can change the timeout period for a specific client session using the **SetClientIdleTime** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

See Also

[SetClientIdleTime](#)

CHttpServer::GetClientLocalPath Method

```
INT GetClientLocalPath(  
    UINT nClientId,  
    LPCTSTR lpszVirtualPath,  
    LPTSTR lpszLocalPath,  
    INT nMaxLength,  
);
```

```
INT GetClientLocalPath(  
    UINT nClientId,  
    LPCTSTR lpszVirtualPath,  
    CString& strLocalPath  
);
```

Return the full local path for a virtual filename or directory on the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszVirtualPath

A pointer to a string that specifies a virtual path on the server. This parameter cannot be NULL.

lpszLocalPath

A pointer to a string buffer that will contain the full local path, terminated with a null-character. This buffer should be at least MAX_PATH characters to accommodate the complete path. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID does not specify a valid client session, the method will return zero. If the string buffer is not large enough to contain the complete path, this method will return zero and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL.

Remarks

The **GetClientLocalPath** method takes a virtual path and returns the full path to the specified file or directory on the local system. The virtual path may be absolute or relative to the root directory for the client session.

To obtain the virtual path for a local file or directory, use the **GetClientVirtualPath** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientVirtualPath](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetClientServer Method

```
HSERVER HttpGetClientServer(  
    UINT nClientId  
);
```

The **GetClientServer** method returns a handle to the server that created the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the method succeeds, the return value is the handle to the server that created the client session. If the method fails, the return value is `INVALID_SERVER`. To get extended error information, call the **GetLastError** method.

Remarks

The **GetClientServer** method returns the handle to the server that created the client session and is typically used within a notification message handler. If the server is in the process of shutting down, or the client session thread is terminating, this method will fail and return `INVALID_SERVER` indicating that the session ID is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cstools10.h`.

Import Library: `cshtsv10.lib`

See Also

[AsyncNotify](#)

CHttpServer::GetClientThreadId Method

```
DWORD GetClientThreadId(  
    UINT nClientId  
);
```

Returns the thread ID associated with the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the method succeeds, the return value is a thread ID. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **GetClientThreadId** method returns a thread ID that can be used to identify the thread that is managing the client session. The thread ID can be used with other Windows API functions such as **OpenThread**. Exercise caution when using thread-related functions, interfering with the normal operation of the thread can have unexpected results. You should never use this method to obtain a thread handle and then call the **TerminateThread** function to terminate a client session. This will prevent the thread from releasing the resources that were allocated for the session and can leave the server in an unstable state. To terminate a client session, use the **DisconnectClient** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[EnumClients](#), [GetActiveClient](#)

CHttpServer::GetClientUserName Method

```
INT GetClientUserName(  
    UINT nClientId,  
    LPTSTR lpszUserName,  
    INT nMaxLength  
);
```

```
INT GetClientUserName(  
    UINT nClientId,  
    CString& strUserName  
);
```

Return the user name associated with the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

strUserName

A pointer to a string buffer that will contain the user name associated with the client session. This buffer must be large enough to store the complete user name, including the terminating null character. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, or the client has not authenticated itself, the method will return zero and the *lpszUserName* parameter will be set to an empty string.

The **IsClientAuthenticated** method can be used to determine if the client has provided credentials as part of the request made to the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AuthenticateClient](#), [GetClientAccess](#)

CHttpServer::GetClientVariable Method

```
INT GetClientVariable(  
    UINT nClientId,  
    LPCTSTR lpszName,  
    LPTSTR lpszValue,  
    INT nMaxLength,  
);
```

```
INT GetClientVariable(  
    UINT nClientId,  
    LPCTSTR lpszName,  
    CString& strValue  
);
```

Return the value of a CGI environment variable for the specified client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszName

A pointer to a string that specifies the name of the environment variable. Variable names are not case-sensitive and should not include the equal sign which acts as a delimiter that separates the variable name from its value. This parameter cannot be a NULL pointer or an empty string.

lpszValue

A pointer to a buffer that will contain the value of the environment variable, terminated with a null character. To determine the length of the header value, this parameter can be NULL and the *nMaxLength* parameter should be specified with a value of zero.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the value buffer, including the terminating null character. If the *lpszValue* parameter is NULL, this value must be zero.

Return Value

If the method succeeds, the return value is the length of the environment variable value, not including the terminating null character. If the client ID does not specify a valid client session, or there is no environment variable that matches the given name, the method will return zero. If the *lpszValue* parameter is not NULL and the buffer is not large enough to store the complete header value, the method will return zero and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL. If the method fails, the **GetLastError** method will return more information about the last error that has occurred.

Remarks

The **GetClientVariable** method will return the value of an environment variable that has been defined for the client. Each client session inherits a copy of the process environment block, which is then modified to define various environment variables that are used with CGI programs and scripts. The **SetClientVariable** method can be used to change existing environment variables or create new variables.

The standard CGI environment variables that are defined by the server are not created until the client request has been processed. This means that environment variables such as REMOTE_ADDR

and SERVER_NAME will not be defined inside an **OnConnect** event handler.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SetClientVariable](#)

CHttpServer::GetClientVirtualHost Method

```
INT GetClientVirtualHost(  
    UINT nClientId,  
    LPTSTR lpszHostName,  
    INT nMaxLength  
);  
  
INT GetClientVirtualHost(  
    UINT nClientId,  
    CString& strHostName  
);
```

Return the name of the virtual host the client used to establish the connection.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszHostName

A pointer to a string buffer that will contain the virtual host name. The string buffer will be null terminated and must be large enough to store the complete hostname. If this parameter is NULL, the method will only return the length of the current command in characters, not including the terminating null character. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. If the maximum length specified is smaller than the actual length of the parameter, this method will fail. If the *lpszHostName* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code.

Remarks

The **GetClientVirtualHost** method is used to obtain the hostname that the client used to establish a connection with the server. This method is typically used within an event handler to determine the hostname associated with the request made by the client. It should not be called inside an **OnConnect** event handler because the virtual host has not been selected at that point. If the virtual hostname is not available at the time this method is called, the method will return zero and the last error code will be set to ST_ERROR_VIRTUAL_HOST_NOT_FOUND.

The **GetClientVirtualHostId** method can be used to obtain the virtual host ID associated with the hostname.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientVirtualHostId](#), [GetCommandUrl](#), [GetVirtualHostName](#)

CHttpServer::GetClientVirtualHostId Method

```
UINT GetClientVirtualHostId(  
    UINT nClientId  
);
```

Return the virtual host ID associated with the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

An unsigned integer value which specifies the virtual host ID. If the method fails, the return value will be `INVALID_VIRTUAL_HOST` and the **GetLastError** method can be used to retrieve the last error code.

Remarks

The **GetClientVirtualHostId** method is used to obtain the virtual host ID associated with the hostname the client used to establish a connection with the server. This method should not be called inside an **OnConnect** event handler because the virtual host has not been selected at that point. If the virtual host ID is not available at the time this method is called, the method will return `INVALID_VIRTUAL_HOST` and the last error code will be set to `ST_ERROR_VIRTUAL_HOST_NOT_FOUND`.

The **GetClientVirtualHost** method can be used to obtain the hostname that the client used to establish the connection.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshtsv10.lib`

See Also

[GetClientVirtualHost](#), [GetCommandUrl](#), [GetVirtualHostId](#)

CHttpServer::GetClientVirtualPath Method

```
INT GetClientVirtualPath(  
    UINT nClientId,  
    LPCTSTR lpszLocalPath,  
    LPTSTR lpszVirtualPath,  
    INT nMaxLength,  
);
```

```
INT GetClientVirtualPath(  
    UINT nClientId,  
    LPCTSTR lpszLocalPath,  
    CString& strVirtualPath  
);
```

Return the virtual path for a local file on the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszLocalPath

A pointer to a string that specifies an absolute path on the local system. This parameter cannot be NULL.

lpszVirtualPath

A pointer to a string buffer that will contain the virtual path, terminated with a null-character. This buffer should be at least MAX_PATH characters to accommodate the complete path. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID does not specify a valid client session, the method will return zero. If the string buffer is not large enough to contain the complete path, this method will return zero and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL.

Remarks

A virtual path for the client is relative to the root directory for the specified client session. These virtual paths are what the client will see as an absolute path on the server. For example, if the server was configured to use "C:\ProgramData\MyServer" as the root directory, and the *lpszLocalPath* parameter was specified as "C:\ProgramData\MyServer\Documents\Research", this method would return the virtual path to that directory as "/Documents/Research".

If the *lpszLocalPath* parameter specifies a file or directory outside of the server root directory, this method will return zero and the last error code will be set to ST_ERROR_INVALID_FILE_NAME. This method can only be used with authenticated clients. If the *nClientId* parameter specifies a client session that has not been authenticated, this method will return zero and the last error code will be ST_ERROR_AUTHENTICATION_REQUIRED.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetCommandFile](#), [GetClientLocalPath](#)

CHttpServer::GetCommandFile Method

```
INT GetCommandFile(  
    UINT nClientId,  
    LPTSTR lpszFileName,  
    INT nMaxLength  
);
```

```
INT GetCommandFile(  
    UINT nClientId,  
    CString& strFileName  
);
```

Get the full path to a file name or directory specified by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A pointer to a string buffer that will contain the full path to a file name or directory specified by the client when it issued a command. The string buffer will be null terminated and must be large enough to store the complete file path. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. It is recommended that the buffer be at least MAX_PATH characters in size. If the maximum length specified is smaller than the actual length of the full path, this method will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code.

Remarks

The **GetCommandFile** method is used to obtain the full path to a local file name or directory specified by the client as an argument to a standard HTTP command. For example, if the client sends the GET command to the server, this method will return the complete path to the local file that the client wants to retrieve. This method will only work with those standard commands that perform some action on a file or directory.

This method should always be used to obtain the file name for a command that performs a file or directory operation. It normalizes the path provided by the client and ensures that it specifies a file or directory name in the correct location. The **GetCommandUrl** method can be used to obtain the URL that was provided by the client.

To map a virtual path to a file or directory on the local system, use the **AddVirtualPath** method. To redirect a client to use a different URL to access the resource, use the **RedirectRequest** method.

The **SetCommandFile** method can be used to change the name of the local file or directory that is the target of the command, however using this method to redirect access to a resource can have unintended side-effects, particularly in the case where the URL provided by the client actually

resolves to an executable CGI program that handles the request.

If the client has provided a URL that resolves to a CGI program that handles the request, this method behaves differently than if the URL is resolved to a local file or directory. If the client uses the GET or POST command that results in a program being executed to handle the request, this method will return the path to the server root directory along with any additional path information provided in the URL. In other words, the file name returned by this method will be the same as the PATH_TRANSLATED value passed to the CGI program.

This method should only be called after the client request has been received by the server, typically inside a **OnCommand** or **OnExecute** event handler. It should not be called inside a **OnConnect** event handler because the server has not processed the client request at that point.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AddVirtualPath](#), [GetCommandUrl](#), [RedirectRequest](#), [SetCommandFile](#), [OnCommand](#), [OnExecute](#)

CHttpServer::GetCommandLine Method

```
INT GetCommandLine(  
    UINT nClientId,  
    LPTSTR lpszCmdLine,  
    INT nMaxLength  
);
```

```
INT GetCommandLine(  
    UINT nClientId,  
    CString& strCmdLine  
);
```

Return the complete command line issued by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCmdLine

A pointer to a string buffer that will contain the command, including all arguments. The string buffer will be null terminated and must be large enough to store the complete command line. If this parameter is NULL, the method will return the length of the command line. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. The internal limit on the maximum length of a command is 1024 characters. If the maximum length specified is smaller than the actual length of the complete command, this method will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code. If the last error code has a value of zero then no command has been issued by the client.

Remarks

The **GetCommandLine** method is used to obtain the command that was issued by the client, and is commonly used inside the **OnCommand** and **OnResult** event handlers to pre-process and post-process client commands, respectively. When the method returns, the string buffer provided by the caller will contain the complete command, including the resource path and requested HTTP version. Any extraneous whitespace will be removed, however all encoding will be preserved.

To obtain the complete URL associated with the request issued by the client, use the **GetCommandUrl** method. If the command sent by the client is used to perform an action on a file or directory, the **GetCommandFile** method should be called to obtain the full path to the local file rather than using the resource path.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetCommandFile](#), [GetCommandUrl](#), [OnCommand](#), [OnResult](#)

CHttpRequest::GetCommandName Method

```
INT GetCommandName
    UINT nClientId,
    LPTSTR lpszCommand,
    INT nMaxLength
);
```

```
INT GetCommandName
    UINT nClientId,
    CString& strCommand
);
```

Return the name of the last command issued by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCommand

A pointer to a string buffer that will contain the command name. The string buffer will be null terminated and must be large enough to store the complete parameter value. If this parameter is NULL, the method will only return the length of the current command in characters, not including the terminating null character. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. If the maximum length specified is smaller than the actual length of the parameter, this method will fail. If the *lpszCommand* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code. If the last error code is returned as a value of zero, this means that no command has been issued by the client.

Remarks

The **GetCommandName** method is used to obtain the name of the last command that was issued by the client. The command name returned by this method will always be capitalized, regardless of how it was sent by the client. This method is typically used inside the **OnCommand** and **OnResult** event handlers to pre-process and post-process client commands, respectively. It should not be called inside a **OnConnect** event handler because the server has not processed the client request at that point.

The **GetCommandUrl** function can be used to return the resource that was requested by the client.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetCommandFile](#), [GetCommandUrl](#), [OnCommand](#), [OnResult](#)

CHttpRequest::GetCommandQuery Method

```
INT GetCommandQuery(  
    UINT nClientId,  
    LPTSTR lpszParameters,  
    INT nMaxLength  
);  
  
INT GetCommandQuery(  
    UINT nClientId,  
    CString& strParameters  
);
```

Return the query parameters included with the command.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszParameters

A pointer to a string buffer that will contain the query parameters when the method returns. The string buffer will be null terminated up to the maximum number of characters specified by the caller. If this parameter is NULL the method will return the length of the query string. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. If the *lpszParameters* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code. If the client did not provide any query parameters, this method will return zero and the last error code will be zero.

Remarks

The **GetCommandQuery** method is used to obtain a copy of the query parameters that were included in the request URL. If there were no query parameters, the string buffer will be empty and the return value will be zero. If the request did include query parameters, they will be returned to the caller in their original, encoded form.

This method should not be called within an **OnConnect** event handler because the client request has not been processed at that point.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshtsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetCommandName](#), [GetCommandResource](#), [ReceiveRequest](#)

CHttpRequest::GetCommandResource Method

```
INT GetCommandResource(  
    UINT nClientId,  
    LPTSTR lpszResource,  
    INT nMaxLength  
);
```

```
INT GetCommandResource(  
    UINT nClientId,  
    CString& strUrlPath  
);
```

Return the URL path for the resource requested by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszResource

A pointer to a string buffer that will contain the URL path provided by the client for the current command. The string buffer will be null terminated and must be large enough to store the complete path. If this parameter is NULL, the method will only return the length of the path. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. If the *lpszResource* parameter is NULL, this value should be zero. If this value is less than the length of the URL, the method will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code.

Remarks

The **GetCommandResource** method returns the URL path that the client provided to access the requested resource. The path will not include the URI scheme, user credentials or query parameters. The **GetCommandFile** method can be used to determine the name of the local file on the server that will be accessed using this URL.

If you require the complete URL, not just the path to the resource, use the **GetCommandUrl** method.

This method should only be called after the client request has been received by the server, typically inside an **OnCommand** event handler. It should not be called inside a **OnConnect** event handler because the server has not processed the client request at that point.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetCommandFile](#), [GetCommandQuery](#), [GetCommandUrl](#), [RedirectRequest](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CHttpServer::GetCommandResult Method

```
INT GetCommandResult(  
    UINT nClientId,  
    LPTSTR lpszResult,  
    INT nMaxLength  
);  
  
INT GetCommandResult(  
    UINT nClientId,  
    CString& strResult  
);
```

Return the result code and description for the last command issued by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszResult

A pointer to a string buffer that will contain the description of the result code. The string buffer will be null terminated up to the maximum number of characters specified by the caller. This parameter can be NULL if this information is not required. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. If the *lpszResult* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the result code for the last command issued by the client. A return value of zero indicates that the command has not completed and there is no result code available.

Remarks

The **GetCommandResult** method is used to determine the result of the last command that was issued by the client and is typically called in the **OnResult** event handler. This method should only be called after a command has been processed or the **SendResponse** method has been called.

The result code is a three-digit integer value that indicates the success or failure of a command. Whenever a client sends a command to the server, the server must respond with this numeric code and a brief description of the the result. Result codes are generally broken down into the following categories:

Result Code	Description
100-199	Result codes in this range are informational and only used with version 1.1 of the protocol. If the server returns a result code in this range, it means that it has received the request and that the client should proceed.
200-299	Result codes in this range indicate that the server has successfully completed the requested action. In most cases this means that the server has returned the requested data to the client, however if a 204 result code is sent, this indicates that the request has been processed but there is no data available.
300-399	Result codes in this range indicate that the requested resource has been moved

	to a new location. The most common result codes are 301 and 302. A value of 301 indicates that the location of the resource has changed permanently and all future requests should be sent to the new URL. A value of 302 indicates that the resource location has changed temporarily. The new location of the resource is sent to the client by setting the Location response header field.
400-499	Result codes in this range indicate an error with the request that was made by the client. These types of errors include invalid commands, requests that can only be issued by authenticated clients, or resources that cannot be accessed on the server. The most common result code in this range is the 404 code which indicates that the requested document could not be found.
500-599	Result codes in this range indicate a server error has occurred while processing a valid request. These types of errors are returned when a command has not been implemented, the execution of a CGI program or script has failed unexpectedly, or an internal server error has occurred.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ReceiveRequest](#), [SendResponse](#), [OnResult](#)

CHttpServer::GetCommandUrl Method

```
INT GetCommandUrl(  
    UINT nClientId,  
    LPTSTR lpszUrl,  
    INT nMaxLength  
);
```

```
INT GetCommandUrl(  
    UINT nClientId,  
    CString& strUrl  
);
```

Return the complete URL of the resource requested by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszUrl

A pointer to a string buffer that will contain the URL provided by the client for the current command. The string buffer will be null terminated and must be large enough to store the complete URL. If this parameter is NULL, the method will only return the length of the URL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. If the *lpszUrl* parameter is NULL, this value should be zero. If this value is less than the length of the URL, the method will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code.

Remarks

The **GetCommandUrl** method returns the complete URL that the client provided to access the requested resource. The URL will include any query parameters that were specified by the client, but it will not include any user credentials. The **GetCommandFile** method can be used to determine the name of the local file on the server that will be accessed using this URL.

If you only require the URL path, without the URI scheme or the query parameters, use the **GetCommandResource** method.

This method should only be called after the client request has been received by the server, typically inside an **OnCommand** event handler. It should not be called inside an **OnConnect** event handler because the server has not processed the client request at that point.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetCommandFile](#), [GetCommandResource](#), [RedirectRequest](#), [OnCommand](#)

CHttpRequest::GetDirectory Method

```
INT GetDirectory(  
    LPTSTR lpszDirectory,  
    INT nMaxLength  
);  
  
INT GetDirectory(  
    CString& strDirectory  
);
```

Return the full path to the root directory assigned to the specified server.

Parameters

lpszDirectory

A pointer to a string buffer that will contain the server root directory, terminated with a null character. It is recommended that this buffer be at least MAX_PATH characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the buffer is not large enough to store the complete path, the method will return a value of zero.

Remarks

The **GetDirectory** method will return the full path to the root directory assigned to the server instance. The root directory may be specified as part of the server configuration, or if no directory is specified by the application, the current working directory for the process will be used and this method can be used to obtain the full path to the directory. When the application specifies a root directory, it may use environment variables such as %AppData% in the path. This method will return the fully resolved path name, with all environment variables expanded.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetAddress](#), [GetIdentity](#), [GetName](#), [SetDirectory](#)

CHttpServer::GetHandle Method

```
HSERVER GetHandle();
```

The **GetHandle** method returns the server handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the server handle associated with the current instance of the class object. If the server is inactive, the value `INVALID_SERVER` will be returned.

Remarks

This method is used to obtain the server handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshtsv10.lib`

See Also

[IsInitialized](#)

CHttpRequest::GetIdentity Method

```
INT GetIdentity(  
    LPTSTR lpszIdentity,  
    INT nMaxLength  
);  
  
INT GetIdentity(  
    CString& strIdentity  
);
```

Return the identity of the specified server.

Parameters

lpszIdentity

A pointer to a string buffer that will contain the identity of the server when the method returns, terminated with a null character. This parameter cannot be NULL. It is recommended that this buffer be at least 32 characters in length. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the buffer is not large enough to store the complete path, the method will return a value of zero.

Remarks

The **GetIdentity** method returns the identity string that was specified as part of the server configuration. It is used for informational purposes only and does not affect the operation of the server. Typically the string specifies the name of the application and a version number. The **SetIdentity** method can be used to change the identity string associated with the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoos10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SetIdentity](#)

CHttpServer::GetLastError Method

```
DWORD GetLastError(  
    LPTSTR lpszError,  
    INT nMaxLength  
);  
  
DWORD GetLastError(  
    CString& strError  
);  
  
DWORD GetLastError();
```

Return the last server error code and a description of the error.

Parameters

lpszError

A pointer to a string buffer that will contain a description of the error. If the error description is not needed, this parameter may be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the error string buffer, including the terminating null character. If the *lpszError* parameter is NULL, this value should be zero.

Return Value

An unsigned integer value that specifies the last error that occurred. A value of zero indicates that there was no error.

Remarks

Error codes are unsigned 32-bit values which are private to each server. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods clear the last error code when they succeed.

It is important to note that the error codes returned by this method are different than the command result codes that are defined in RFC 2616, the standard protocol specification for HTTP. This method is used to determine reason that an API function has failed, and should not be used to determine if a command issued by the client was successful. The **SendResponse** method is used to send responses to the client, and the **GetCommandResult** method can be used to determine the result of the last command sent by the client.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Failure is typically indicated by a return value such as FALSE, NULL, INVALID_SERVER or HTTP_ERROR. Those methods which clear the last error code when they succeed are noted on their reference page.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetCommandResult](#), [SendResponse](#), [SetLastError](#)

CHttpServer::GetLogFile Method

```
BOOL GetLogFile(  
    UINT * lpnLogFormat,  
    UINT * lpnLogLevel,  
    LPTSTR lpszFileName,  
    INT nMaxLength  
);
```

```
BOOL GetLogFile(  
    UINT * lpnLogFormat,  
    UINT * lpnLogLevel,  
    CString& strFileName  
);
```

```
BOOL GetLogFile(  
    CString& strFileName  
);
```

Return the current log file format and the full path to the file.

Parameters

lpnLogFormat

A pointer to an integer value that will contain the log file format being used when the method returns. If this information is not needed, this parameter may be NULL. The following formats are supported:

Constant	Description
HTTP_LOGFILE_NONE (0)	This value specifies that the server should not create or update a log file.
HTTP_LOGFILE_COMMON (1)	This value specifies that the server should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about file transfers.
HTTP_LOGFILE_COMBINED (2)	This value specifies that the server should use the combined log file format. This format is similar to the common format, however it includes the client referrer and user agent. This is the format that most Apache web servers use by default.
HTTP_LOGFILE_EXTENDED (3)	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session.

lpnLogLevel

A pointer to an integer value that will contain the level of detail the server uses when generating the log file. The minimum value is 1 and the maximum value is 10. If this information is not needed, this parameter may be NULL.

lpszFileName

A pointer to a string buffer that will contain the full path to the log file. This parameter may be

NULL if this information is not required. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the file name string, including the terminating null character. If the *lpzFileName* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the current log file format. Refer to the **HTTPSERVERCONFIG** structure definition for a list of supported log file formats. If logging has not been enabled, this method will return a value of zero.

Remarks

If the server is configured with logging enabled, but a log file name is not explicitly provided, then the server will automatically generate one. This method can be used to get the full path to the current log file along with the format that is being used to record client session data. Normally the log file is held open by the server thread while it is active, however you can call the **RenameServerLogFile** method to explicitly rename or delete the log file.

To change the name of the log file, the log file format or level of detail, use the **SetLogFile** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RenameServerLogFile](#), [SetLogFile](#)

GetMemoryUsage Method

```
SIZE_T GetMemoryUsage();
```

Return the amount of memory allocated for the server and all client sessions.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero and specifies the amount of memory allocated by the server. If the server is inactive or cannot be locked, the return value is zero. Call the **GetLastError** method to determine the cause of the failure.

Remarks

This method returns the amount of memory allocated by the server and all active client sessions. It enumerates all of memory allocations made by the server process and client session threads and returns the total number of bytes allocated for the server process. This value reflects the amount of memory explicitly allocated by this library and does not reflect the total working set size of the process, or memory allocated by any other libraries. To determine the working set size for the process, refer to the Win32 **GetProcessWorkingSetSize** and **GetProcessMemoryInfo** functions.

This method forces the server into a locked state, and all client sessions will block until the method returns. Because this method enumerates all heaps allocated for the server process, it can be an expensive operation, particularly when there are a large number of active clients connected to the server. Frequent use of this method can significantly degrade the performance of the server. It is primarily intended for use as a debugging tool to determine if memory usage is the result of an increase in active client sessions. If the value returned by the method remains reasonably constant, but the amount of memory allocated for the process continues to grow, it could indicate a memory leak in some other area of the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetStackSize](#), [SetStackSize](#)

CHttpRequest::GetName Method

```
INT GetName(  
    UINT nClientId,  
    LPTSTR lpszHostName,  
    INT nMaxLength  
);  
  
INT GetName(  
    UINT nClientId,  
    CString& strHostName  
);
```

Return the host name assigned to the specified server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszHostName

A pointer to a string buffer that will contain the server host name, terminated with a null character. It is recommended that this buffer be at least 64 characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the client ID is invalid or the buffer is not large enough to store the complete hostname, the method will return a value of zero.

Remarks

This method will return the host name assigned to the specified server. If the *nClientId* parameter has a value of zero, the method will return the default host name that was specified as part of the server configuration. If no host name was explicitly assigned to the server, then it will return the local system name. If the *nClientId* parameter specifies a client session, then it this method will return the host name that the client used to establish the connection.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetAddress](#)

CHttpServer::GetOptions Method

```
DWORD GetOptions();
```

Return the options specified for this instance of the server.

Parameters

None.

Return Value

The current server options. For a list of available options, see [Server Option Constants](#)

Remarks

The **GetOptions** method returns the default options for the current instance of the server. To change the server options, use the **SetOptions** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

See Also

[SetOptions](#)

CHttpRequest::GetPriority Method

```
INT GetPriority();
```

Return the current priority assigned to the specified server.

Parameters

None.

Return Value

If the method succeeds, the return value is the priority for the specified server. If the method fails, the return value is HTTP_PRIORITY_INVALID. To get extended error information, call the **GetLastError** method.

Remarks

The **GetPriority** method can be used to determine the current priority assigned to the server. It will return one of the following values:

Constant	Description
HTTP_PRIORITY_BACKGROUND (0)	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
HTTP_PRIORITY_LOW (1)	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
HTTP_PRIORITY_NORMAL (2)	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
HTTP_PRIORITY_HIGH (3)	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
HTTP_PRIORITY_CRITICAL (4)	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cstown10.h.

Import Library: cshtsv10.lib

See Also

[SetPriority](#), [Start](#)

CHttpServer::GetProgramExitCode Method

```
BOOL GetProgramExitCode(  
    UINT nClientId,  
    DWORD& dwExitCode  
);
```

Return the exit code of the last program executed by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwExitCode

An unsigned integer that will contain the program exit code when the method returns.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **GetProgramExitCode** method returns the exit code of a registered CGI program or script that was executed. By convention, most programs return an exit code in the range of 0-255, with an exit code of zero indicating success. The exit code is commonly used by programs to communicate status information back to the server application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[GetProgramOutput](#), [RegisterProgram](#)

CHttpServer::GetProgramName Method

```
INT GetProgramName(  
    UINT nClientId,  
    LPTSTR lpszProgramName,  
    INT nMaxLength  
);  
  
INT GetProgramName(  
    UINT nClientId,  
    CString& strProgramName  
);
```

Return the name of the CGI program executed by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszProgramName

A pointer to a string buffer that will contain the name of the CGI program executed by the client. This parameter cannot be NULL and should be at least MAX_PATH characters in size. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This parameter must have a value greater than zero.

Return Value

If the method succeeds, the return value is the length of the executable file name. If the client ID does not specify a valid client session, the method will return zero. If the client has not executed a CGI program this method will return zero and the last error code will be set to zero.

Remarks

The **GetProgramName** method returns the local file name of the CGI program that was executed in response to a request from the client. This is the full path to the executable that was registered with the server using the **RegisterProgram** method. If the client specified a regular document or directory, this method will return a value of zero, indicating that no CGI program has been executed to handle the request.

To obtain the resource URL that was provided by the client, use the **GetCommandUrl** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetProgramExitCode](#), [GetProgramOutput](#), [RegisterProgram](#)

CHttpServer::GetProgramOutput Method

```
DWORD GetProgramOutput(  
    UINT nClientId,  
    LPBYTE lpBuffer,  
    DWORD dwBufferSize  
);
```

Return a copy of the standard output from the a CGI program executed by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpBuffer

A pointer to a buffer that will contain the output from the last program executed by the client. If this parameter is NULL, the method will return the number of bytes of data that was output by the program. Note that this output is not null terminated.

dwBufferSize

The maximum number of bytes that can be copied into the buffer. If the *lpBuffer* parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of bytes copied into the specified buffer. If the client ID does not specify a valid client session, the method will return zero. If the client has not executed any programs, the return value will be zero.

Remarks

The **GetProgramOutput** method is used to obtain a copy of the output generated by a CGI program. To determine the number of bytes of output available to read, call this method with the *lpBuffer* parameter as NULL and the *dwBufferSize* parameter with a value of zero. The return value will be the number of bytes of data that was output by the program. It should be noted that for Unicode builds, the buffer is a byte array, not an array of characters, and will not be null terminated.

This method returns the raw output from the program which may contain a response header block, escape sequences, control characters and embedded nulls. When the application processes the output returned by this method, it should never coerce the buffer pointer to an LPTSTR value because there is no guarantee that the data will be null-terminated. To obtain the output from the program as a null-terminated string, use the **GetProgramText** method.

This method should only be used within an **OnExecute** event handler, which occurs after the program has terminated.

Example

```
LPBYTE lpBuffer = NULL; // A pointer to the output buffer  
DWORD cbBuffer = 0;     // Number of bytes in the output buffer  
  
// Determine the number of bytes in the output buffer  
cbBuffer = pHttpServer->GetProgramOutput(nClientId, NULL, 0);  
  
if (cbBuffer > 0)  
{
```

```
// Allocate memory for the buffer
lpBuffer = new BYTE[cbBuffer + 1];

// Copy the program output to the buffer
cbBuffer = pHttpServer->GetProgramOutput(nClientId, lpBuffer, cbBuffer + 1);
}

// Free the memory allocated for the buffer when finished
if (lpBuffer != NULL)
{
    delete lpBuffer;
    lpBuffer = NULL;
    cbBuffer = 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[GetProgramExitCode](#), [GetProgramText](#), [OnExecute](#), [RegisterProgram](#)

CHttpServer::GetProgramText Method

```
INT GetProgramText(  
    UINT nClientId,  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);
```

```
INT GetProgramText(  
    UINT nClientId,  
    CString& strBuffer  
);
```

Return a copy of the standard output from a CGI program in a string buffer.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszBuffer

A pointer to a buffer that will contain the output from the last program executed by the client as a string. If this parameter is NULL, the method will return the number of bytes of characters that was output by the program, not including a terminating null character. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

The maximum number of bytes that can be copied into the buffer. If the *lpszBuffer* parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of characters copied into the specified string buffer, not including the terminating null character. If the client ID does not specify a valid client session, the method will return zero. If the client has not executed any programs, the return value will be zero.

Remarks

The **GetProgramText** method is used to obtain a copy of the output generated by a CGI program. To determine the number of characters of output available to read, call this method with the *lpszBuffer* parameter as NULL and the *nMaxLength* parameter with a value of zero. The return value will be the number of characters that were output by the program. If the application dynamically allocates the string buffer, make sure that it allocates an extra character for the terminating null character.

This method will only return textual output from the program and any non-printable control characters and the escape character will be replaced with a space. To obtain the unfiltered output from the program, use the **GetProgramOutput** method. If the program outputs a response header block, this will be included in the string buffer.

This method should only be used within an **OnExecute** event handler, which occurs after the program has terminated.

Example

```
CString strBuffer;
```

```
if (pHttpServer->GetProgramText(nClientId, strBuffer) > 0)
    pEditCtrl->SetWindowText(strBuffer);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetProgramExitCode](#), [GetProgramOutput](#), [OnExecute](#), [RegisterProgram](#)

CHttpRequest::GetStackSize Method

```
DWORD GetStackSize();
```

Return the initial size of the stack allocated for threads created by the server.

Parameters

None.

Return Value

If the method succeeds, the return value is the amount of memory that will be allocated for the stack in bytes. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetStackSize** method returns the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes.

If this method is called when the server is not active, it will return the default stack size that the server will be configured to use when it starts. In this case, a return value of zero specifies that the default stack size will be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cstools10.h`.

Import Library: `cshtsv10.lib`

See Also

[GetMemoryUsage](#), [SetStackSize](#), [Start](#)

CHttpServer::GetTransferInfo Method

```
BOOL GetTransferInfo(  
    UINT nClientId,  
    LPHTTPSERVERTRANSFER lpTransferInfo  
);
```

Return information about the current file transfer.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpTransferInfo

A pointer to an **HTTPSERVERTRANSFER** structure that will contain information about the last file transfer. This parameter cannot be NULL, and the *dwSize* member of the structure must be initialized to specify the structure size prior to calling this method.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero. This method should only be called after the client has issued the GET or PUT commands to initiate a file transfer, otherwise the return value will be zero.

Remarks

The **GetTransferInfo** method is used to obtain information about the last file transfer that was performed by the client. This method is typically called within an event handler to determine how many bytes of data were transferred, the type of file and the full path to the file on the local system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ReceveRequest](#), [SendResponse](#), [HTTPSERVERTRANSFER](#)

CHttpServer::GetUuid Method

```
INT GetUuid(  
    LPTSTR lpszHostUuid,  
    INT nMaxLength  
);  
  
INT GetUuidString(  
    CString& strHostUuid  
);
```

Return the UUID assigned to the server as a printable string.

Parameters

lpszHostUuid

A pointer to a string buffer that will contain the server UUID, terminated with a null character. It is recommended that this buffer be at least 40 characters in length. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the buffer is not large enough to store the complete UUID string, the method will return a value of zero.

Remarks

The **GetUuid** method returns the Universally Unique Identifier (UUID) that has been assigned to the server. The UUID may either be generated by the application and assigned as part of the server configuration, or an ephemeral UUID may be automatically generated when the server is started.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SetUuid](#)

CHttpServer::GetVirtualHostId Method

```
UINT GetVirtualHostId(  
    LPCTSTR lpszHostName,  
    UINT nHostPort  
);
```

Return the virtual host ID associated with the specified hostname.

Parameters

lpszHostName

A string that specifies the virtual host name.

nHostPort

An optional integer value which specifies the port number for the virtual host. If this parameter is specified it must be the same value as the original port number that the server was configured to use.

Return Value

If the method succeeds, the return value is the host ID that uniquely identifies the virtual host. If the server handle is invalid, or there is no virtual host with the specified name, the method will return VIRTUAL_HOST_UNKNOWN. If the method fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **GetVirtualHostId** method is used to obtain the unique virtual host ID that is associated with a specific hostname. This method will match both the primary virtual hostname added using the **AddVirtualHost** method, as well as any aliases that were added using the **AddVirtualHostAlias** method. To obtain the virtual host ID associated with the active client session, use the **GetClientVirtualHostId** method.

The *nHostPort* parameter should either be omitted or specified with the same port number that the server was configured to use. Port-based virtual hosting is currently not supported and this parameter is included for future use.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AddVirtualHost](#), [AddVirtualHostAlias](#), [DeleteVirtualHost](#), [GetClientVirtualHostId](#), [GetVirtualHostName](#)

CHttpServer::GetVirtualHostName Method

```
INT GetVirtualHostName(  
    UINT nHostId,  
    LPTSTR lpszHostName,  
    INT nMaxLength  
);
```

```
INT GetVirtualHostName(  
    UINT nHostId,  
    CString& strHostName  
);
```

Return the hostname associated with the specified virtual host ID.

Parameters

hServer

The server handle.

nHostId

An integer value which identifies the virtual host.

lpszHostName

A pointer to a string buffer that will contain the virtual host name, terminated with a null character. It is recommended that this buffer be at least 64 characters in length. If this parameter is NULL, the method will return the length of the virtual hostname. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. If the *lpszHostName* parameter is NULL this value must be zero.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the server handle or the host ID is invalid, or the buffer is not large enough to store the complete hostname, the method will return a value of zero.

Remarks

The **GetVirtualHostName** method returns the primary hostname associated with the specified virtual host ID. This is the same hostname that was specified when the virtual host was added to the server configuration using the **AddVirtualHost** method. To obtain the hostname that was used by the active client session to connect to the server, use the **GetClientVirtualHost** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

CHttpServer::IsActive Method

```
BOOL IsActive();
```

Determine if the server has been started.

Return Value

This method returns a non-zero value if the server has been started. If the server is stopped this method will return zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[CHttpServer](#), [IsListening](#), [Start](#), [Stop](#)

CHttpServer::IsClientAuthenticated Method

```
BOOL IsClientAuthenticated(  
    UINT nClientId  
);
```

Determine if the specified client session has been authenticated.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the client session has been authenticated, this method will return a non-zero value, otherwise it will return zero. If the client ID is valid, this method will clear the last error code.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[AuthenticateClient](#), [GetClientCredentials](#), [RequireAuthentication](#)

CHttpServer::IsCommandEnabled Method

```
BOOL IsCommandEnabled(  
    LPCTSTR lpszCommand  
);
```

Determine if a specific server command has been enabled or disabled.

Parameters

lpszCommand

A pointer to a NULL terminated string that specifies the name of the command. The command name is not case-sensitive, but the value must otherwise match the exact command name.

Partial matches are not recognized by this method. This parameter cannot be NULL.

Return Value

If the command is enabled, this method will return a non-zero value. If the command is disabled or the command name does not match a supported command, this method will return zero.

Remarks

The **IsCommandEnabled** method is used to determine whether a specific command is enabled. Typically this method is used in an event handler to make sure the command issued by a client is recognized by the server and enabled for use. Commands can be enabled using the **EnableCommand** method and disabled using the **DisableCommand** method.

This method does not account for the permissions granted to a specific client session. Clients are assigned access rights when they are authenticated using the **AuthenticateClient** method, and certain commands can be limited by the permissions granted to the client. For example, even if the PUT command is enabled, a client must have the HTTP_ACCESS_WRITE permission to use the command to upload a file to the server. For a list of access rights, see [User Access Constants](#).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AuthenticateClient](#), [DisableCommand](#), [EnableCommand](#), [GetCommandName](#)

CHttpRequest::IsInitialized Method

BOOL IsInitialized();

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[CHttpServer](#)

CHttpRequest::IsListening Method

```
BOOL IsListening();
```

Determine if the server is listening for client connections.

Return Value

This method returns a non-zero value if the server has been started and is listening for client connections. If the server is stopped or has been suspended this method will return zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

See Also

[CHttpServer](#), [IsActive](#), [Start](#), [Stop](#)

CHttpServer::PreProcessEvent Method

```
virtual LONG PreProcessEvent(  
    HSERVER hServer,  
    UINT nClientId,  
    UINT nEventId,  
    DWORD dwError,  
    BOOL& bHandled  
);
```

A virtual method that is invoked for each event generated by the server.

Parameters

hServer

The server handle. The application should treat this as an opaque value that is only valid as long as the server is active. This value should not be stored by the application and the handle value will change if the server is restarted.

nClientId

An unsigned integer which uniquely identifies the client that has issued a request to the server. This value is guaranteed to be unique to the client session throughout the life of the server and is never reused. The application should never make assumptions about the order in which IDs are allocated to the client sessions.

nEventId

An unsigned integer which specifies which event occurred. For a list of events, see [Server Event Constants](#).

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

bHandled

An integer which specifies if the event has been handled by the application. If this parameter is set to a non-zero value, the default event handler will not be invoked for the event.

Return Value

The method should return a value of zero to indicate that the default event handler should be invoked for the event. If the method returns a non-zero value, this value is passed back to the event dispatcher and the default handler will not be invoked.

Remarks

The **PreProcessEvent** method is invoked for each event that is generated, prior to the default handler for that event. To implement an event handler, the application should create a class derived from the **CHttpServer** class, and then override this method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

CHttpRequest::ReceiveRequest Method

```
BOOL ReceiveRequest(  
    UINT nClientId,  
    DWORD dwOptions,  
    LPHTTPREQUEST LpRequest,  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength  
);
```

Receive the request that was sent by the client to the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwOptions

An unsigned integer which specifies how the client request data will be copied. It may be one of the following values:

Constant	Description
HTTP_REQUEST_DEFAULT (0)	If the <i>lpdwLength</i> parameter points to a integer with a non-zero value, the <i>lpvBuffer</i> parameter is considered to be a pointer to a block of memory that has been allocated to store the request data. This is the same as specifying the HTTP_REQUEST_MEMORY option. If the <i>lpdwLength</i> parameter points to an integer with a value of zero, the <i>lpvBuffer</i> parameter is considered to be a pointer to an HGLOBAL memory handle. This is the same as specifying the HTTP_REQUEST_HGLOBAL option.
HTTP_REQUEST_MEMORY (0x1)	The <i>lpvBuffer</i> parameter is a pointer to a block of memory that has been allocated to store the request data. The maximum number of bytes of data that can stored is determined by the value of the integer that the <i>lpdwLength</i> parameter points to. When the method returns, that value will updated with with actual number of bytes copied into the buffer.
HTTP_REQUEST_STRING (0x2)	The <i>lpvBuffer</i> parameter is a pointer to a string buffer that has been allocated to store the request data. The maximum number of bytes of data that can stored is determined by the value of the integer that the <i>lpdwLength</i> parameter points to. When the method returns, that value will updated with with actual number of bytes copied into the buffer.
HTTP_REQUEST_HGLOBAL (0x4)	The <i>lpvBuffer</i> parameter is a pointer to an HGLOBAL memory handle. When the method returns, the handle will reference a block of memory that contains the request data submitted by the client. The <i>lpdwLength</i> parameter will contain the number of bytes copied to the buffer.

HTTP_REQUEST_FILE (0x8)	The <i>lpvBuffer</i> parameter is a pointer to a string which specifies the name of a file that will contain the request data. If the file does not exist, it will be created. If it does exist, the contents will be replaced. This option is typically used in conjunction with the PUT command. If the <i>lpdwLength</i> parameter is not NULL, the value it points to will be updated with the actual number of bytes stored in the file.
HTTP_REQUEST_HANDLE (0x10)	The <i>lpvBuffer</i> parameter is a handle to an open file. This option is typically used in conjunction with the POST or PUT commands. If the <i>lpdwLength</i> parameter is not NULL, the value it points to will be updated with the actual number of bytes written to the file. If this option is specified, the request data will be written from the current position in the file and will advance the file pointer by the number of bytes received from the client.

lpRequest

A pointer to a **HTTPREQUEST** structure which contains information about the request from the client. This parameter cannot be NULL. The structure that is passed to this method must have all members set to a value of zero except the *dwSize* member, which must be initialized to the size of the structure.

lpvBuffer

A pointer to the buffer that will contain any request data that was submitted by the client. The *dwOptions* parameter determines if this pointer references a block of memory, a null-terminated string buffer, a global memory handle or a file name. If this parameter is NULL, any data submitted by the client will not be copied.

lpdwLength

A pointer to an unsigned integer that will contain the number of bytes of data submitted by the client when the method returns. If the *lpvBuffer* parameter specifies a memory or string buffer, this value must be initialized to the maximum size of the buffer before the method is called. If the *lpvBuffer* parameter points to a global memory handle, this value must be initialized to zero. If *lpvBuffer* is NULL or specifies a file name, this parameter may be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **ReceiveRequest** method is called within an **OnCommand** event handler to process the command issued by the client and return information about the request to the server application. It is only necessary for the application to call this method if it wants to implement its own custom handling for a command. It is recommended that most applications use the default command processing for standard commands such as GET and POST to ensure that the appropriate security checks are performed and the response conforms to the protocol standard.

This method may only be called once per command issued by the client and the data referenced in the **HTTPREQUEST** structure only remains valid while the client is connected to the server. You should never attempt to directly modify the data referenced by any of the structure members. If you wish to store or modify any of the string values returned in the structure, you should allocate

a buffer large enough to store the contents of the string, including the terminating null character, and copy the string into that buffer.

If the HTTP_REQUEST_HGLOBAL option is used to return a copy of the request data in a global memory buffer, the HGLOBAL handle must be freed by the application when the data is no longer needed. Failure to free this handle will result in a memory leak.

If the HTTP_REQUEST_HANDLE option is used to write a copy of the request data to an open file, the handle must reference a disk file that was opened or created using the **CreateFile** function with GENERIC_WRITE access. It cannot be a handle to a device or named pipe. If the method succeeds, the file pointer is advanced by the number of bytes of request data submitted by the client. If the method fails, the file pointer is returned to its original position prior to the method being called.

Example

```
// Initialize the HTTPREQUEST structure
HTTPREQUEST httpRequest;
ZeroMemory(&httpRequest, sizeof(httpRequest));
httpRequest.dwSize = sizeof(httpRequest);

// Return the data in a global memory buffer
HGLOBAL hglobalBuffer = NULL;
DWORD dwLength = 0;

bSuccess = pServer->ReceiveRequest(nClientId,
                                   HTTP_REQUEST_HGLOBAL,
                                   &httpRequest,
                                   &hglobalBuffer,
                                   &dwLength);

if (bSuccess && hglobalBuffer != NULL)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hglobalBuffer);

    if (lpBuffer != NULL)
    {
        // Process dwLength bytes of data submitted by the client
    }

    GlobalUnlock(hglobalBuffer);
    GlobalFree(hglobalBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SendResponse](#), [HTTPREQUEST](#)

CHttpServer::RedirectRequest Method

```
BOOL RedirectRequest(  
    UINT nClientId,  
    UINT nMethod,  
    LPCTSTR lpszLocation  
);
```

Redirect the request from the client to another URL.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nMethod

An integer value that specifies if the redirection is permanent or temporary. The following values may be used:

Constant	Description
HTTP_REDIRECT_PERMANENT (1)	This value is used for permanent redirection, indicating that the client should update any record of the link with the new URL specified by the <i>lpszLocation</i> parameter. This result is cacheable and when the client makes subsequent requests for the resource, it should always use the new URL.
HTTP_REDIRECT_TEMPORARY (2)	This value is used for temporary redirection, indicating that the client should issue a request for the resource using the new URL specified by the <i>lpszLocation</i> parameter, but subsequent requests should continue to use the original URL.
HTTP_REDIRECT_OTHER (3)	This value is used for temporary redirection, however it instructs the client that it should use the GET command to request the redirected resource. This option is typically used to redirect a client after it has used the POST command.

lpszLocation

A pointer to a string that specifies the new location for the requested resource. This value must be a complete URL, including the http:// or https:// scheme. This parameter cannot be NULL or point to zero-length string.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **RedirectRequest** method can be used within a HTTP_CLIENT_COMMAND event handler to redirect the client to a new location for the resource that it has requested. This redirection can be permanent or temporary, depending on whether the server expects the client to continue to use the original URL when requesting the resource.

If the `HTTP_REDIRECT_TEMPORARY` method is used, the actual status code that is returned to the client depends on the version of the protocol that is being used. If the client has issued the request using HTTP 1.0 then the server will return a 302 code to the client. If the client is using HTTP 1.1, the server will return a 307 code to the client that indicates it should use the same command verb (GET, POST, etc.) when requesting the resource at the new location.

If the `HTTP_REDIRECT_OTHER` method is used, the status code that is returned to the client depends on which version of the protocol is being used. For clients who are using HTTP 1.0, the server will return a 302 code to the client just as with the `HTTP_REDIRECT_TEMPORARY` method. If the client is using HTTP 1.1, the server will return a 303 code to the client that indicates it should always use the GET command to request the new resource, regardless if a different command was originally used (POST, PUT, etc.)

This method provides a simplified interface for sending a redirection status code that also implicitly sets the Location response header to the value of the *lpszLocation* parameter. If the server application needs to send alternate redirection codes such as 305 (Use Proxy) then it should use the **SendReponse** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshtsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RequireAuthentication](#), [SendErrorResponse](#), [SendResponse](#)

CHttpRequest::RegisterHandler Method

```
BOOL RegisterHandler(  
    UINT nHostId,  
    LPCTSTR lpszExtension,  
    LPCTSTR lpszProgramFile,  
    LPCTSTR lpszParameters,  
    LPCTSTR lpszDirectory  
);
```

Register a CGI program for use and associate it with a file name extension.

Parameters

nHostId

An unsigned integer that identifies the virtual host associated with the program. The value VIRTUAL_HOST_DEFAULT should be used for the default host that is created when the server is first started.

lpszExtension

A pointer to a string which specifies the file name extension that is associated with the CGI program. This parameter cannot be NULL.

lpszProgramFile

A pointer to a string that specifies the full path to the executable program. This parameter cannot be NULL.

lpszParameters

A pointer to a string that specifies additional parameters for the program. This value will be passed to the program as command line arguments. If the program does not require any command line parameters, this value may be NULL or point to an empty string.

lpszDirectory

A pointer to a string that specifies the current working directory for the program. If this parameter is NULL or points to an empty string, the server will use the root document directory for the virtual host.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **RegisterHandler** method registers an executable CGI program and associates it with a file name extension. When the client issues a GET or POST command that specifies a file with that extension, the program will be executed and the output return to the client.

The *lpszProgramFile* string specifies file name of the CGI program. You should not install any executable programs in the server root directory or its subdirectories. A client should never have the ability to directly access the executable file itself. It is permitted to have multiple file name extensions that reference the same program. The only requirement is that the extension be unique for the given host. The program name may contain environment variables surrounded by % symbols. For example, %ProgramFiles% would be expanded to the **C:\Program Files** folder.

It is important to note that the program specified by *lpszProgramFile* must be an executable file, not a script or batch file. If the program name does not contain a directory path, then the

standard Windows pathing rules will be used when searching for an executable file that matches the given name. It is recommended that you always provide a full path to the executable file.

The *lpszParameters* string can specify additional command line parameters that should be passed to the CGI program as arguments. This string can also contain a placeholder named "%1" that will be replaced by the full path to the local script filename. If no placeholder is included in the parameters, or *lpszParameters* is a NULL pointer, the script file name will be passed to the program as its only argument.

The executable program that is registered using this program must be a console application that conforms to the CGI/1.1 specification defined in RFC 3875. Request data submitted by the client as part of a POST will be provided to the program as standard input. The output from the program must be written to standard output. The first lines of output from the program should be any response headers, followed by an empty line. Each line should be terminated with a carriage-return and linefeed (CRLF) sequence. If the CGI program outputs additional data to be processed by the client, it should provide Content-Type and Content-Length response headers.

The application can obtain a copy of the output from the command by calling the **GetProgramOutput** method from within an **OnExecute** event handler.

When developing a CGI program, it is important to take into consideration the environment that it will be executing in. The program will be started as a child process of the server application, and will inherit the same privileges. This means that it will typically have access to the boot drive, the Windows folders and the system registry. CGI programs must ensure that all query parameters and request data submitted by the client have been validated.

If the server is running on a system with User Account Control (UAC) enabled and does not have elevated privileges, do not register a program that requires elevated privileges or has a manifest that specifies the requestedExecutionLevel as requiring administrative privileges.

Example

```
// Register a handler for VBScript
pServer->RegisterHandler(hServer,
                        VIRTUAL_HOST_DEFAULT,
                        _T("vbs"),
                        _T("%SystemRoot%\System32\cscript.exe"),
                        _T("/nologo /b \"%1\""),
                        NULL);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetProgramExitCode](#), [GetProgramOutput](#), [RegisterProgram](#)

CHttpServer::RegisterProgram Method

```
BOOL RegisterProgram(  
    LPCTSTR lpszVirtualPath,  
    LPCTSTR lpszProgramFile,  
);
```

```
BOOL RegisterProgram(  
    LPCTSTR lpszVirtualPath,  
    LPCTSTR lpszProgramFile,  
    LPCTSTR lpszParameters,  
    LPCTSTR lpszDirectory  
);
```

Register a CGI program for use and associate it with a virtual path on the server.

Parameters

nHostId

An unsigned integer that identifies the virtual host associated with the program. The value VIRTUAL_HOST_DEFAULT should be used for the default host that is created when the server is first started.

lpszVirtualPath

A pointer to a string which specifies the virtual path to the CGI program. This must be an absolute path, but does not have to specify a pre-existing virtual path or map to the directory structure of the root document directory for the server. This parameter cannot be NULL. The maximum length of the virtual path is 1024 characters.

lpszProgramFile

A pointer to a string that specifies the full path to the executable program. This parameter cannot be NULL.

lpszParameters

A pointer to a string that specifies additional parameters for the program. This value will be passed to the program as command line arguments. If the program does not require any command line parameters, this value may be NULL or point to an empty string.

lpszDirectory

A pointer to a string that specifies the current working directory for the program. If this parameter is NULL or points to an empty string, the server will use the root document directory for the virtual host.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value will be zero and the **GetLastError** method can be used to retrieve the last error code.

Remarks

The **RegisterProgram** function registers an executable CGI program and associates it with a virtual path. When the client issues a GET or POST command specifying the virtual path associated with the program, the program will be executed and the output return to the client.

The *lpszProgramFile* string specifies file name of the CGI program. You should not install any executable programs in the server root directory or its subdirectories. A client should never have the ability to directly access the executable file itself. It is permitted to have multiple virtual paths

that reference the same executable file. The only requirement is that the virtual path be unique for the given host. The program name may contain environment variables surrounded by % symbols. For example, %ProgramFiles% would be expanded to the **C:\Program Files** folder.

It is important to note that the program specified by *lpzProgramFile* must be an executable file, not a script or batch file. If the program name does not contain a directory path, then the standard Windows pathing rules will be used when searching for an executable file that matches the given name. It is recommended that you always provide a full path to the executable file.

The *lpzParameters* string can specify additional command line parameters that should be passed to the CGI program as arguments. This string can also contain a placeholder named "%1" that will be replaced by the virtual path associated with the program. If *lpzParameters* is NULL or a zero-length string, then no additional parameters are passed to the program.

The executable program that is registered using this program must be a console application that conforms to the CGI/1.1 specification defined in RFC 3875. Request data submitted by the client as part of a POST will be provided to the program as standard input. The output from the program must be written to standard output. The first lines of output from the program should be any response headers, followed by an empty line. Each line should be terminated with a carriage-return and linefeed (CRLF) sequence. If the CGI program outputs additional data to be processed by the client, it should provide Content-Type and Content-Length response headers.

The application can obtain a copy of the output from the command by calling the **GetProgramOutput** function from within an **OnExecute** event handler.

When developing a CGI program, it is important to take into consideration the environment that it will be executing in. The program will be started as a child process of the server application, and will inherit the same privileges. This means that it will typically have access to the boot drive, the Windows folders and the system registry. CGI programs must ensure that all query parameters and request data submitted by the client have been validated.

If the server is running on a system with User Account Control (UAC) enabled and does not have elevated privileges, do not register a program that requires elevated privileges or has a manifest that specifies the requestedExecutionLevel as requiring administrative privileges.

Example

```
m_pServer->RegisterProgram(hServer,
                           VIRTUAL_HOST_DEFAULT,
                           HTTP_METHOD_DEFAULT,
                           _T("/order/invoice"),
                           _T("%ProgramData%\MyServer\Programs\invoice.exe"),
                           NULL,
                           NULL);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoos10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetProgramExitCode](#), [GetProgramOutput](#), [RegisterHandler](#)

CHttpRequest::RenameServerLogFile Method

```
BOOL RenameServerLogFile(  
    LPCTSTR lpszFileName  
);
```

Rename or delete the current log file being updated by the server.

Parameters

lpszFileName

A pointer to a string that specifies the file name the current log file should be renamed to. If this parameter is NULL or an empty string, the current log file will be deleted.

Return Value

If the method succeeds, the return value is non-zero. If logging is not currently enabled for the server, this method will return zero.

Remarks

The **RenameServerLogFile** method is used to rename or delete the current log file. Note that this does not change the current log file name or disable logging by the server. It only changes the file name of the current log file, or removes the log file if the *lpszFileName* parameter is NULL. This can be useful if you want your server to perform log file rotation, archiving the current log file. By renaming the current log file, the server will automatically create a new log file with original file name.

This method must be used to rename or delete the current log file while logging is active because the server holds an open handle on the file. The application should not use the **GetLogFile** method to obtain the log file name and then use the **MoveFileEx** or **DeleteFile** Windows API functions with that file.

To disable logging, use the **SetLogFile** method and specify the logging format as HTTP_LOGFILE_NONE.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLogFile](#), [SetLogFile](#)

CHttpServer::RequireAuthentication Method

```
BOOL RequireAuthentication(  
    UINT nClientId,  
    UINT nAuthType,  
    LPCTSTR lpszRealm  
);
```

Send a response to the client indicating that authentication is required.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nAuthType

An integer value that corresponds to a result code, informing the client if the redirection is permanent or temporary. The following values may be used:

Constant	Description
HTTP_AUTH_BASIC (1)	This option specifies the Basic authentication scheme should be used. This option is supported by all clients that support at least version 1.0 of the protocol.

lpszRealm

A pointer to a string that is displayed by a web browser to indicate to the user which username and password they should use. If this parameter is NULL or an empty string, the domain name the client used to establish the connection will be used.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **RequireAuthentication** method can be used within an **OnCommand** event handler to indicate to the client that it must provide a username and password to access the requested resource. The client should respond by issuing another request that includes the required credentials. To determine if a client has included credentials with its request, use the **IsClientAuthenticated** method. The **GetClientCredentials** method will return the username and password that was provided by the client.

Some clients may require that the session be secure if authentication is requested or display warning messages to the user if the connection is not secure. It is recommended that you enable security using the HTTP_OPTION_SECURE option if your application will require clients to authenticate before accessing specific resources.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientCredentials](#), [IsClientAuthenticated](#), [SendResponse](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CHttpServer::Restart Method

```
BOOL Restart();
```

Restart the server, terminating all active client sessions.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Restart** method will restart the specified server, terminating all active client sessions. If the method is unable to restart the server for any reason, the server thread is terminated. The server retains all of the configuration parameters from the previous instance, however the statistical information (such as the number of clients, files transferred, etc.) will be reset.

If an application calls this method from within an event handler, the active client session (the client for which the event handler was invoked) may not get a disconnect notification. It is recommended that this method only be called by the same thread that created the server using the **Start** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[Start](#), [Stop](#)

CHttpServer::Resume Method

BOOL Resume();

Resume accepting client connections.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Resume** method instructs the server to resume accepting new client connections after the **Suspend** method has been called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[Restart](#), [Start](#), [Stop](#), [Suspend](#), [Throttle](#)

CHttpRequest::SendErrorResponse Method

```
BOOL SendErrorResponse(  
    UINT nClientId,  
    UINT nErrorCode,  
    LPCTSTR lpszMessage  
);
```

Send a customized error response to the specified client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nErrorCode

An integer value that specifies the error code that should be sent to the client. This value should correspond to the error result codes defined for HTTP in RFC 2616, which are three-digit values in the range of 400 through 599. The method will fail if an invalid error code is specified.

lpszMessage

A pointer to a string that describes the error. If this parameter is NULL or specifies a zero-length string, a default message will be selected based on the error code.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **SendErrorResponse** method sends a response to the client indicating that an error has occurred, providing a numeric error code and HTML formatted text which may be displayed to the user. The *lpszMessage* parameter should provide a brief description of the error that will be included in the output sent to the client. Note that the message should not contain any special formatting control characters or HTML markup.

This method provides a simplified interface for sending an error response to the client. In some cases, a browser may choose to display its own error message to the user in place of the generic HTML document generated by this method. If you want your application to send a customized HTML document for a specific type of error, you should use the **SendResponse** method.

If you wish to redirect the client to use an alternate URL to access the requested resource, it is recommended that you use the **RedirectRequest** method rather than sending an error response.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RedirectRequest](#), [SendResponse](#)

CHttpServer::SendResponse Method

```
BOOL SendResponse(  
    UINT nClientId,  
    DWORD dwOptions,  
    LPHTTPRESPONSE lpResponse,  
    LPVOID lpvBuffer,  
    DWORD dwLength  
);
```

Send a response from the server to the specified client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwOptions

An unsigned integer which specifies how the client request data will be copied. It may be one of the following values:

Constant	Description
HTTP_RESPONSE_DEFAULT (0)	The <i>lpvBuffer</i> parameter is a pointer to a block of memory that contains the response data. The number of bytes of data that in the buffer is specified by the <i>dwLength</i> parameter. This is the same as specifying the HTTP_RESPONSE_MEMORY option.
HTTP_RESPONSE_MEMORY (0x1)	The <i>lpvBuffer</i> parameter is a pointer to a block of memory that contains the response data. The number of bytes of data that in the buffer is specified by the <i>dwLength</i> parameter. The data will be sent to the client as a stream of bytes. If the server application was compiled using Unicode, it is responsibility of the application to convert any Unicode text to either ANSI or UTF-8, depending on the resource that was requested by the client.
HTTP_RESPONSE_STRING (0x2)	The <i>lpvBuffer</i> parameter is a pointer to a string buffer that contains the response data. The maximum number of bytes of data that will be sent to the client is determined by the <i>dwLength</i> parameter. If the value of the <i>dwLength</i> parameter exceeds the string length, the value will be ignored and the contents of the string will be sent up to the terminating null character. If the Unicode version of this method is called, the string will be converted to a byte array before being sent to the client.
HTTP_RESPONSE_HGLOBAL (0x4)	The <i>lpvBuffer</i> parameter is an HGLOBAL memory handle which references a block of memory that contains the response data. The number of bytes of data that in the buffer is specified by the <i>dwLength</i>

	parameter. The data will be sent to the client as a stream of bytes. It is the responsibility of the application to free the global memory handle after it is no longer needed.
HTTP_RESPONSE_FILE (0x8)	The <i>lpvBuffer</i> parameter is a pointer to a string which specifies the name of a file that contains the response data. If the file does not exist, or does not specify a regular file, this method will fail. The <i>dwLength</i> parameter is ignored. If the content type for the specified file is not explicitly defined in the response, the method will attempt to automatically determine the correct type based on the file name extension and/or the contents of the file.
HTTP_RESPONSE_HANDLE (0x10)	The <i>lpvBuffer</i> parameter is a handle to an open file and the <i>dwLength</i> parameter specifies the number of bytes to be read from the file and send to the client. If this option is specified, the response data will be read from the current position in the file and will advance the file pointer by the number of bytes sent to the client.
HTTP_RESPONSE_DYNAMIC (0x0010)	The response data will be generated dynamically. This prevents the content length from being included in the response header, and forces the connection to close, regardless if the keep-alive option has been specified. This option must be specified if the application wishes to use the SendResponseData method to send additional data to the client.
HTTP_RESPONSE_NOCACHE (0x0020)	Informs the client that the data being returned by the server should not be cached. Typically this is used in conjunction with the HTTP_RESPONSE_DYNAMIC option when the data is being generated dynamically.

lpResponse

A pointer to a **HTTPRESPONSE** structure which contains additional information about the response to the client. The structure that is passed by reference to this method must have the *dwSize* member initialized to the size of the structure or the method will fail. This parameter may be NULL, in which case a default response of "200 OK" is sent to the client along with any data specified by the *lpvBuffer* parameter.

lpvBuffer

A pointer to the buffer that will contain any response data that should be sent to the client. The *dwOptions* parameter determines if this pointer references a block of memory, a null-terminated string buffer, a global memory handle or a file name. This parameter may be NULL, in which case no data will be sent to the client. If this method is called in response to a HEAD command being sent by the client, this parameter is ignored.

dwLength

An unsigned integer that specifies the number of bytes of data to be sent to the client. If the *lpvBuffer* parameter is NULL, this value must be zero. If this method is called in response to a HEAD command being sent by the client, this parameter is ignored.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **SendResponse** method is called within an **OnCommand** event handler to respond to the request made by the client. This method may only be called once after a command has been received, and must be called after the **ReceiveRequest** method. It is only necessary for the application to call this method if it wants to implement its own custom handling for a command. It is recommended that most applications use the default command processing for standard commands such as GET and POST to ensure that the appropriate security checks are performed and the response conforms to the protocol standard.

If the HTTP_RESPONSE_HANDLE option is used to read a copy of the response data from an open file, the handle must reference a disk file that was opened using the **CreateFile** function with GENERIC_READ access. It cannot be a handle to a device or named pipe. If the *dwLength* parameter is larger than the total number of bytes available to be read from the current position in the file, the method will stop sending data to the client when it reaches the end-of-file. If the method succeeds, the file pointer is advanced by the number of bytes of response data sent to the the client. If the method fails, the file pointer is returned to its original position prior to the method being called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RecieveRequest](#), [SendResponseData](#), [OnCommand](#), [HTTPRESPONSE](#)

CHttpRequest::SendResponseData Method

```
INT SendResponseData(  
    UINT nClientId,  
    LPBYTE lpBuffer,  
    INT nLength  
);
```

Send additional data to the client in response to a command.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpBuffer

A pointer to a buffer that contains the response data that should be sent to the client. This parameter cannot be NULL.

nLength

An integer that specifies the number of bytes of data in the buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes of data sent to the client. If the method fails, the return value is HTTP_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

The **SendResponseData** method is called within a **OnCommand** event handler to send data to the client in response to a request. This method can only be used to send dynamically generated content after the **SendResponse** method has been called. The HTTP_RESPONSE_DYNAMIC option must have been specified when responding to the client or this method will fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RecieveRequest](#), [SendResponse](#), [OnCommand](#), [HTTPRESPONSE](#)

CHttpServer::SetCertificate Method

```
BOOL SetCertificate(  
    DWORD dwProtocol,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName,  
    LPCTSTR lpszPassword  
);
```

```
BOOL SetCertificate(  
    LPCTSTR lpszCertName,  
    LPCTSTR lpszPassword  
);
```

Set the name of the certificate to be used with secure connections.

Parameters

dwProtocol

An unsigned integer that specifies the security protocols to be used when establishing a secure connection with the client. This parameter may be one or more of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default selection of security protocols will be used when establishing a connection. The TLS 1.2, TLS 1.1 and TLS 1.0 protocols will be negotiated with the server, in that order of preference. This option will always request the latest version of the preferred security protocols and is the recommended value.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Note that SSL 2.0 has been deprecated and will never be used unless the server does not support version 3.0.
SECURITY_PROTOCOL_TLS	The TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.

lpszCertStore

A pointer to a string which specifies the name of certificate store. This may be the name of the certificate store in registry, or it may specify the name of a file that contains the certificate and

its private key.

lpzCertName

A pointer to a string which specifies the common name for the certificate that will be used. Typically this will be the fully qualified domain name for the server.

lpzPassword

An optional pointer to a string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only required if the *lpzCertStore* parameter specifies a certificate file in PKCS12 format that is password protected.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **SetCertificate** method will create the security credentials required for the server to accept secure connections and enable security options for the server. This method will not validate the certificate information provided by the application. If the certificate does not exist, or does not have a private key associated with it, the client will be unable to establish a secure connection.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetOptions](#), [SetOptions](#)

CHttpServer::SetClientAccess Method

```
BOOL SetClientAccess(  
    UINT nClientId,  
    DWORD dwUserAccess  
);
```

Change the access rights associated with the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies one or more user access rights. For a list of user access rights that can be granted to the client, see [User and File Access Constants](#).

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero. This method can only be used with authenticated clients. If the client session has not been authenticated, the return value will be zero.

Remarks

The **SetClientAccess** method can change multiple access rights for the client session. The **EnableClientAccess** method can be used to grant or revoke a specific permission for the client session.

If the *dwUserAccess* parameter has a value of HTTP_ACCESS_DEFAULT, then default permissions will be granted to the client session based on the configuration of the server. This is the recommended value for most clients. It is important to consider the implications of changing the access permissions granted to a client session. For example, if you do not grant clients HTTP_ACCESS_READ permission, it can effectively disable the site because the server will return 403 Forbidden errors for all GET and HEAD requests.

This function should typically be called in the **OnConnect** event handler to assign general permissions to the client, or in the **OnCommand** event handler after the client has issued a request and provided any authentication credentials that are required.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: cshtsv10.lib

See Also

[AuthenticateClient](#), [EnableClientAccess](#), [GetClientAccess](#)

CHttpServer::SetClientHeader Method

```
BOOL HttpSetClientHeader(  
    UINT nClientId,  
    UINT nHeaderType,  
    LPCTSTR lpszHeaderName,  
    LPCTSTR lpszHeaderValue  
);
```

Create or change the value of a request or response header for the client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nHeaderType

Specifies the type of header to create or modify. It may be one of the following values:

Constant	Description
HTTP_HEADERS_REQUEST	Change a request header that was provided by the client. Request header values provide additional information to the server about the type of request being made.
HTTP_HEADERS_RESPONSE	Change a response header that was created by the server. Response header values provide additional information to the client about the type of information that is being returned by the server.

lpszHeaderName

A pointer to a string that specifies the name of the header that should be created or modified. Header names are not case-sensitive and should not include the colon which acts as a delimiter that separates the header name from its value. This parameter cannot be a NULL pointer or an empty string.

lpszHeaderValue

A pointer to a string that specifies the new value of the header. If this parameter is a NULL pointer or an empty string, it has the same effect as deleting the header value from the list of request or response headers.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero. If the method fails, the **GetLastError** method will return more information about the last error that has occurred.

Remarks

The **SetClientHeader** method will change the value of a request or response header for the specified client session. If the *lpszHeaderName* value matches an existing header field, its value will be replaced. If the header name is not defined, then a new header will be created with the given value. You should not change the value of most standard response header values unless you are certain of the impact that it would have on the normal operation of the client.

If you wish to define a custom header value that would be included in the response to a client

request, you should prefix the header name with "X-" to avoid potential conflicts with the standard response headers. For example, if you wanted to identify a customer, you could create a header field with the name "X-Customer-ID" and set the value to the customer ID number. The client application would receive this custom header value as part of the response to its request for a document.

Refer to [Hypertext Transfer Protocol Headers](#) for a list of common request and response headers that are used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DeleteClientHeader](#), [GetClientHeader](#)

CHttpServer::SetClientIdleTime Method

```
UINT SetClientIdleTime(  
    UINT nClientId,  
    UINT nTimeout  
);
```

Change the idle timeout period for the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nTimeout

An unsigned integer value that specifies the number of seconds that the client may remain idle. If this value is zero, the default idle timeout period for the server will be used.

Return Value

If the method succeeds, the return value is the previous client idle timeout period in seconds. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **SetClientIdleTime** method will change the number of seconds that the client may remain idle before being automatically disconnected by the server. The minimum timeout period for a client is 10 seconds, the maximum is 300 seconds (5 minutes). The idle time of a client session is based on the last time a command was issued to the server or when a data transfer completed.

If the value INFINITE is specified as the timeout period, the client activity timer will be refreshed, extending the idle timeout period for the session. This is typically done inside an **OnTimeout** event handler to prevent the client from being disconnected due to inactivity.

To obtain the current idle timeout period for a client, along with the amount of time the client has been idle, use the **GetClientIdleTime** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[GetClientIdleTime](#)

CHttpServer::SetClientVariable Method

```
BOOL HttpSetClientVariable(  
    UINT nClientId,  
    LPCTSTR lpszName,  
    LPCTSTR lpszValue  
);
```

Create or change the value of a CGI environment variable for the specified client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszName

A pointer to a string that specifies the name of the environment variable that should be created or modified. Environment variables are not case-sensitive and should not include the equal sign which acts as a delimiter that separates the variable name from its value. This parameter cannot be a NULL pointer or an empty string.

lpszValue

A pointer to a string that specifies the new value of the environment variable. If this parameter is a NULL pointer or an empty string, it has the same effect as deleting the variable from the environment block for the client session.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero. If the method fails, the **GetLastError** method will return more information about the last error that has occurred.

Remarks

The **SetClientVariable** method will change the value of a environment variable for the specified client session. If the *lpszName* value matches an existing variable, its value will be replaced. If the variable is not defined, then a new variable will be created with the given value. The value of an environment variable can be obtained using the **GetClientVariable** method.

The server will automatically create a number of different environment variables that will be passed to a program or script executed by the server. These variables are defined in RFC 3875 as part of the Common Gateway Interface (CGI) 1.1 specification. The following variables are defined by the server and should not be modified directly by the application:

Variable Name	Description
AUTH_TYPE	The authorization scheme used by the server to authenticate the client session
CONTENT_LENGTH	The length of the request data provided by the client
CONTENT_TYPE	The MIME type that identifies the type of content provided by the client
DOCUMENT_ROOT	The full path to the local document root directory on the server
GATEWAY_INTERFACE	The version of the Common Gateway Interface that is being used by the server

PATH_INFO	The resource or sub-resource that is to be returned by the program or script
PATH_TRANSLATED	The path information mapped to the server root document directory structure
QUERY_STRING	The URL encoded query parameters passed to the program or script
REMOTE_ADDR	The network address of the client sending the request to the server
REMOTE_HOST	The same value as the REMOTE_ADDR variable
REMOTE_USER	The username specified as part of the authentication credentials provided by the client
REQUEST_METHOD	The method used by the client to request the resource
REQUEST_URI	The URI for the script provided by the client
SCRIPT_FILENAME	The full path to the program or script on the server
SCRIPT_NAME	The path to the program or script specified by the client
SERVER_NAME	The hostname or IP address of the server that the client connected to
SERVER_PORT	The port number that the client used to connect to the server
SERVER_PORT_SECURE	This variable has a value of "1" if the client connection to the server is secure
SERVER_PROTOCOL	The version of the server protocol used
SERVER_SOFTWARE	The server identity string which specifies the application name and version

In addition to the environment variables listed, the server will also create variables that are prefixed with "HTTP_" that are set to the value of request headers that are not otherwise defined. For example, the HTTP_USER_AGENT variable will be set to the value of the User-Agent header provided by the client as part of the request.

Calling the **SetClientVariable** method within an **OnExecute** event handler will have no effect because it occurs after the CGI program or script has completed execution. To create or modify environment variables for the client session, it should be done within an **OnCommand** event handler.

This method will not change the environment block for the server process. Each client session is allocated its own private environment block which is inherited by the CGI program. When the client session terminates, the memory allocated for its environment is released.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientVariable](#)

CHttpServer::SetCommandFile Method

```
BOOL SetCommandFile(  
    UINT nClientId,  
    LPCTSTR lpszFileName  
);
```

Change the name of a file or directory that is the target of the current command.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A pointer to a string that specifies the new file name. This parameter may be NULL to specify that the original file or directory name should be used.

Return Value

If the method succeeds, the return value is non-zero. If the client ID does not specify a valid client session, the method will return zero.

Remarks

The **SetCommandFile** method is used by the application to change the target file or directory name for the current command from within an **OnCommand** event handler. This can be used to effectively redirect the client to use a different file than the one that was actually requested. For example, if the client issues the GET command to download a file from the server, this method can be used to redirect the command to use a different file name. To obtain the full path to the file or directory that is the target of the current command, use the **GetCommandFile** method.

The *lpszFileName* parameter specifies the path to the new file or directory name. If the path is absolute, then it will be used as-is. If the path is relative, it will be relative to the root directory for the client session. Because the root document directory for the client can depend on the hostname that the client used when connecting to the server, it is recommended that you always use an absolute path. The full path to this file is not limited to the server root directory or its subdirectory, it can specify a file anywhere on the local system. If this parameter is a NULL pointer, or points to an empty string, then the server will revert to using the actual file or directory name specified by the command. This enables the application to effectively undo a previous call to this method to change the target file name.

Typically this method would be used to redirect a client to a file or directory that it may not normally have access to. Exercise caution when using this method to provide access to data that is stored outside of the server root directory. Incorrect use of this method could expose the server to security risks or cause unpredictable behavior by client applications. In most cases it is preferable to use the **AddVirtualPath** method to create a virtual path or file name on the server, or the **RedirectRequest** method to request the client use a different URL to access the resource.

This method should only be called within the context of the **OnCommand** event, and only for those commands that perform an action on a file or directory. If the current command does not target a file or directory, this method will return zero and the last error code will be set to `ST_ERROR_INVALID_COMMAND`.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AddVirtualPath](#), [GetCommandFile](#), [ReceiveRequest](#), [RedirectRequest](#), [SendResponse](#)

CHttpServer::SetDirectory Method

```
BOOL SetDirectory(  
    LPCTSTR lpszDirectory  
);
```

Specify the local directory that will be used as the server root directory.

Parameters

lpszDirectory

A pointer to a string that specifies the root directory for the server. If this parameter is NULL or a zero-length string, the server will use the current working directory as the root directory.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return zero.

Remarks

The **SetDirectory** method specifies the path to the local directory that should be used as the root document directory for the server.

You cannot change the server root directory after the server has started. To change the root directory, you must stop the server using the **Stop** method and then start another instance of the server with a configuration that specifies the new directory.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetDirectory](#)

CHttpServer::SetIdentity Method

```
BOOL SetIdentity(  
    LPCTSTR lpszIdentity  
);
```

Change the identity of the specified server.

Parameters

lpszIdentity

A pointer to a string that identifies the server. If this parameter is NULL or specifies an empty string, the current identity for the server is reset to a default value. The maximum length of the identity string is 64 characters, including the terminating null character.

Return Value

If the method succeeds, the return value is non-zero, otherwise it will return a value of zero.

Remarks

The **SetClientIdentity** method changes a string value used by the server to identify itself to clients. The identity string does not have any standard format and is used for informational purposes only. Typically it consists of the application name and a version number. Changing the server identity has no effect on the operation of the server. To obtain the identity string currently associated with the server, use the **GetIdentity** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetIdentity](#)

CHttpServer::SetLastError Method

```
VOID SetLastError(  
    DWORD dwError  
);
```

Set the last error code for the specified server session.

Parameters

dwError

An unsigned integer that specifies an error code.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each server session. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SERVER or HTTP_ERROR.

If the *dwError* parameter is specified with a value of zero, this effectively clears the error code for the last method that failed. Those methods which clear the last error code when they succeed are noted on their reference page.

Applications can retrieve the value saved by this method by calling the **GetLastError** method to determine the specific reason for failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

See Also

[GetCommandResult](#), [GetLastError](#), [SendResponse](#)

CHttpServer::SetLogFile Method

```
BOOL SetLogFile(  
    UINT nLogFormat,  
    UINT nLogLevel,  
    LPCTSTR lpszFileName  
);
```

Change the current log format, level of detail and file name.

Parameters

nLogFormat

An integer value that specifies the format used when creating or updating the server log file. The following formats are supported:

Constant	Description
HTTP_LOGFILE_NONE (0)	This value specifies that the server should not create or update a log file.
HTTP_LOGFILE_COMMON (1)	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about GET, PUT and POST requests.
HTTP_LOGFILE_COMBINED (2)	This value specifies that the server should use the combined log file format. This format is similar to the common format, however it includes the client referrer and user agent. This is the format that most Apache web servers use by default.
HTTP_LOGFILE_EXTENDED (3)	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session. This format typically generates the largest logfiles.

nLogLevel

An integer value that specifies the level of detail that should be generated in the log file. The minimum value is 1 and the maximum value is 10. If this parameter is zero, it is the same as specifying a log file format of HTTP_LOGFILE_NONE and will disable logging by the server.

lpszFileName

A pointer to a string that specifies the name of the log file that should be created or appended to. If the server was configured with logging enabled and this parameter is NULL or an empty string, the current log file name will not be changed. If the log file does not exist, it will be created. If it does exist, the contents of the log file will be appended to.

Return Value

If the method succeeds, the return value is non-zero. If one of the parameters are invalid, the method will return zero.

Remarks

The **SetLogFile** method can be used to change the current log file name, the format of the log file

or the level of detail recorded in the log file. In some situations it may be desirable to delete the current log file contents when changing the format or ensure that a new log file is created. To do this, combine the *nLogFormat* parameter with the constant HTTP_LOGFILE_DELETE.

The higher the value of the *nLogLevel* parameter, the greater the level of detail that is recorded by the server. A log level of 1 instructs the server to only record GET, POST and PUT requests, while a level of 10 instructs the server to record all commands processed by the server. Because a higher level of logging detail can negatively impact the performance of the server, it is recommended that you do not exceed a level of 5 for most applications. A log level of 10 should only be used for debugging purposes.

Example

```
UINT nLogFormat = HTTP_LOGFILE_NONE;
UINT nLogLevel = 0;
UINT nNewLevel = 5;
BOOL bChanged = FALSE;

// Change the level of detail for the current log file if logging
// has been enabled and the current level is a lower value

if (pHttpServer->GetLogFile(&nLogFormat, &nLogLevel, NULL, 0))
{
    if (nLogFormat != HTTP_LOGFILE_NONE && nLogLevel < nNewLevel)
        bChanged = pHttpServer->SetLogFile(nLogFormat, nNewLevel, NULL);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLogFile](#), [RenameServerLogFile](#)

CHttpServer::SetName Method

```
BOOL SetName(  
    UINT nClientId,  
    LPCTSTR lpszHostName  
);
```

Change the host name assigned to the specified server or client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszHostName

A pointer to a string that specifies the new host name assigned to the server or client session. If this value is NULL or points to an empty string, the current host name will be changed to use the default host name.

Return Value

If the method succeeds, the return value is non-zero. If the client ID is invalid, or the buffer is not large enough to store the complete hostname, the method will return a value of zero.

Remarks

This method will change the host name assigned to the specified client session. If the *nClientId* parameter has a value of zero, the method will change default host name that was assigned to the server as part of the server configuration. If the *nClientId* parameter specifies a valid client session and the *lpszHostName* parameter is NULL, the host name associated with the client session will be changed to the current host name assigned to the server.

When a client connects to the server, it can specify the host name that it used to establish the connection by sending the HOST command. This is typically used with virtual hosting, where one server may accept client connections for multiple domains. The **GetName** method will return the host name specified by the client, and **SetName** can be used by the application to either explicitly assign a different host name to the client session, or override the host name provided by the client.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetAddress](#), [GetName](#)

CHttpServer::SetOptions Method

```
BOOL SetOptions(  
    DWORD dwOptions  
);
```

Sets the default options for this instance of the server.

Parameters

dwOptions

An unsigned integer which specifies one or more options. For a list of available options, see [Server Option Constants](#).

Return Value

If the method is successful, it will return a non-zero value, otherwise it will return a value of zero.

Remarks

The **SetOptions** method changes the default options for the current instance of the server. This method cannot be used to change the options for an active instance of the server. If the server is active, it must be stopped before calling this method. To get the current options, use the **GetOptions** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

See Also

[GetOptions](#)

CHttpServer::SetPriority Method

```
INT SetPriority(  
    INT nPriority  
);
```

Change the priority assigned to the specified server.

Parameters

nPriority

An integer value which specifies the new priority for the server. It may be one of the following values:

Constant	Description
HTTP_PRIORITY_BACKGROUND (0)	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
HTTP_PRIORITY_LOW (1)	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
HTTP_PRIORITY_NORMAL (2)	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
HTTP_PRIORITY_HIGH (3)	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
HTTP_PRIORITY_CRITICAL (4)	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Return Value

If the method succeeds, the return value is the previous priority assigned to the server. If the method fails, the return value is HTTP_PRIORITY_INVALID. To get extended error information, call the **GetLastError** method.

Remarks

The **SetPriority** method can be used to change the current priority assigned to the specified server. Client connections that are accepted after this method is called will inherit the new priority as their default priority. Previously existing client connections will not be affected by this method.

Higher priority values increase the thread priority and processor utilization for each client session. You should only change the server priority if you understand the impact it will have on the system and have thoroughly tested your application. Configuring the server to run with a higher priority can have a negative effect on the performance of other programs running on the system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cstools10.h`.

Import Library: `cshtsv10.lib`

See Also

[GetPriority](#), [Start](#)

CHttpRequest::SetStackSize Method

```
BOOL SetStackSize(  
    DWORD dwStackSize  
);
```

Change the initial size of the stack allocated for threads created by the server.

Parameters

dwStackSize

The amount of memory that will be committed to the stack for each thread created by the server. If this value is zero, a default stack size will be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetStackSize** method changes the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes. Increasing or decreasing the stack size will only affect new threads that are created by the server, it will not affect those threads that have already been created to manage active client sessions. It is recommended that most applications use the default stack size.

You should not change the stack size unless you understand the impact that it will have on your system and have thoroughly tested your application. Increasing the initial commit size of the stack will remove pages from the total system commit limit, and every page of memory that is reserved for stack cannot be used for any other purpose.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `csstools10.h`

Import Library: `cshtsv10.lib`

See Also

[GetMemoryUsage](#), [GetStackSize](#), [Start](#)

CHttpServer::SetUuid Method

```
BOOL SetUuid(  
    LPCTSTR lpszHostUuid  
);
```

Assign a UUID to the current instance of the server.

Parameters

lpszHostUuid

A pointer to a string that specifies the server UUID, terminated with a null character. This value can be used when storing information about the server, and should be generated using a utility such as **uuidgen** which is included with Visual Studio. This parameter may be NULL or point to an empty string, in which case a temporary UUID will be randomly generated for the server.

Return Value

If the method succeeds, the return value is non-zero, otherwise the method return a value of zero.

Remarks

The **SetUuid** method assigns a Universally Unique Identifier (UUID) to the server. The UUID may either be generated by the application and assigned as part of the server configuration, or an ephemeral UUID may be automatically generated when the server is started. This method cannot be used to change the UUID after the server has been started. To determine the UUID assigned to an active server instance, use the **GetUuid** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetUuid](#)

CHttpServer::Start Method

```
BOOL Start(  
    LPCTSTR lpszLocalHost,  
    UINT nLocalPort,  
    DWORD dwOptions  
);
```

```
BOOL Start(  
    LPCTSTR lpszLocalHost,  
    UINT nLocalPort  
);
```

```
BOOL Start(  
    UINT nLocalPort  
);
```

The **Start** method begins listening for client connections on the specified local address and port number. The server is started in its own thread and manages the client sessions independently of the calling thread. All interaction with the server and its client sessions takes place inside the class event handlers.

Parameters

lpszLocalHost

A pointer to a string which specifies the local hostname or IP address address that the server should be bound to. If this parameter is omitted or specifies a NULL pointer an appropriate address will automatically be used. If a specific address is used, the server will only accept client connections on the network interface that is bound to that address.

nLocalPort

The port number the server should use to accept client connections. If a value of zero is specified, the server will use the standard port number 21 to listen for connections, or port 990 if the server is configured to use implicit SSL. The port number used by the application must be unique and multiple instances of a server cannot use the same port number. It is recommended that a port number greater than 5000 be used for private, application-specific implementations.

dwOptions

An unsigned integer value that specifies one or more options to be used when creating an instance of the server. For a list of the available options, see [Server Option Constants](#). If this parameter is omitted, the default options for the server instance will be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

In most cases, the *lpszLocalHost* parameter should be omitted or a NULL pointer. On a multihomed system, this will enable the server to accept connections on any appropriately configured network adapter. Specifying a hostname or IP address will limit client connections to that particular address. Note that the hostname or address must be one that is assigned to the local system, otherwise an error will occur.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the method will fail.

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

The handle returned by this method references the listening socket that was created when the server was started. The service is managed in another thread, and all interaction with the server and active client connections are performed inside the event handlers. To disconnect all active connections, close the listening socket and terminate the server thread, call the **Stop** method.

The host UUID that is defined as part of the server configuration should be generated using the **uuidgen** utility that is included with the Windows SDK. You should not use the UUID that is provided in the example code, it is for demonstration purposes only. If no host UUID is specified in the server configuration, an ephemeral UUID will be generated automatically when the server is started.

Example

```
CHttpRequest *pHttpRequest = new CHttpRequest();

// Initialize the server configuration
pHttpRequest->SetName(_T("server.company.com"));
pHttpRequest->SetUuid(_T("10000000-1000-1000-1000-100000000000"));
pHttpRequest->SetDirectory(_T("%ProgramData%\MyProgram\Server"));
pHttpRequest->SetLogFile(HTTP_LOGFILE_EXTENDED, 5,
    _T("%ProgramData%\MyProgram\Server.log"));
pHttpRequest->SetOptions(HTTP_SERVER_DEFAULT);

// Start the server
pHttpRequest->Start(HTTP_PORT_DEFAULT);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnumClients](#), [Restart](#), [Stop](#)

CHttpServer::Stop Method

BOOL Stop();

Stop the server, terminating all active client sessions and releasing the resources that were allocated for the server.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero.

Remarks

The **Stop** method instructs the server to stop accepting client connections, disconnects all active client connections and terminates the thread that is managing the server session. The handle is no longer valid after the server has been stopped and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new server is started. An application should always consider the server handle to be opaque and never depend on it being a specific value.

If an application calls this method from within an event handler, the active client session (the client for which the event handler was invoked) may not get a disconnect notification. It is recommended that this method only be called by the same thread that created the server using the **Start** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[Restart](#), [Start](#)

CHttpRequest::Suspend Method

```
BOOL Suspend();
```

Suspend the server and reject new client connections.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Suspend** method instructs the server to suspend accepting new client connections. Any incoming client connections will be rejected with an error message indicating that the server is currently unavailable. To resume accepting client connections, call the **Resume** method. Suspending the server will have no effect on clients that have already established a connection with the server.

It is recommended that you only suspend a server if absolutely necessary, and only for brief periods of time. If you want to limit the number of active client connections or control the connection rate for clients, use the **Throttle** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

See Also

[Restart](#), [Resume](#), [Start](#), [Stop](#), [Throttle](#)

CHttpServer::Throttle Method

```
BOOL Throttle(  
    UINT nMaxClients,  
    UINT nMaxClientsPerAddress,  
    DWORD dwConnectionRate  
);
```

The **Throttle** method limits the number of active client connections, connections per address and connection rate.

Parameters

nMaxClients

A value which specifies the maximum number of clients that may connect to the server. A value of zero specifies that there is no fixed limit to the number of client connections.

nMaxClientsPerAddress

A value which specifies the maximum number of clients that may connect to the server from the same IP address. A value of zero specifies that there is no fixed limit to the number of client connections per address. By default, there is a limit of four client connections per address.

dwConnectionRate

A value which specifies a restriction on the rate of client connections, limiting the number of connections that will be accepted within that period of time. A value of zero specifies that there is no restriction on the rate of client connections. The higher this value, the fewer the number of connections that will be accepted within a specific period of time. By default, there is no limit on the client connection rate.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Throttle** method is used to limit the number of connections and the connection rate to minimize the potential impact of a large number of client connections over a short period of time. This can be used to protect the server from a client application that is malfunctioning or a deliberate denial-of-service attack in which the attacker attempts to flood the server with connection attempts.

If the maximum number of client connections or maximum number of connections per address is exceeded, the server will reject subsequent connection attempts until the number of active client sessions drops below the specified threshold. Note that adjusting these values lower than the current connection limits will not affect clients that have already connected to the server. For example, if the **Start** method is called with the maximum number of clients set to 100, and then **Throttle** is called lowering that value to 75, no existing client connections will be affected by the change. However, the server will not accept any new connections until the number of active clients drops below 75.

Increasing the connection rate value will force the server to slow down the rate at which it will accept incoming client connection requests. For example, setting this parameter to a value of 1000 would limit the server to accepting one client connection every second, while a value of 250 would allow the server to accept four client connections per second. Note that significantly increasing the amount of time the server must wait to accept client connections can exceed the connection

backlog queue, resulting in client connections being rejected.

It is recommended that you always specify conservative connection limits for your server application based on expected usage. Allowing an unlimited number of client connections can potentially expose the system to denial-of-service attacks and should never be done for servers that are accessible over the Internet.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[Restart](#), [Resume](#), [Start](#), [Suspend](#)

CHttpServer Event Handlers

Method	Description
OnAuthenticate	The client requested authentication
OnConnect	The client established a control connection to the server
OnCommand	The client issued a command to the server
OnDisconnect	The client has disconnected from the server
OnDownload	The client has downloaded a file from the server
OnError	The event handler encountered an error when processing a client event
OnExecute	The client has executed an external program on the server
OnResult	The command issued by the client has been processed by the server
OnTimeout	The client has exceeded the maximum allowed idle time
OnUpload	The client has uploaded a file to the server

CHttpServer::OnAuthenticate Method

```
virtual void OnAuthenticate(  
    UINT nClientId,  
    LPCTSTR lpszHostName,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword  
);
```

A virtual method that is invoked after the client has successfully downloaded a file from the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszHostName

A pointer to a string that specifies the host name that the client used to establish the connection.

lpszUserName

A pointer to a string that specifies the user name provided by the client.

lpszPassword

A pointer to a string that specifies the cleartext user password provided by the client.

Return Value

None.

Remarks

The **OnAuthenticate** event handler is invoked when the client has provided authentication credentials as part of the request for a document or other resource. To implement an event handler, the application should create a class derived from the **CHttpServer** class, and then override this method.

The event handler can call the **AuthenticateClient** method to authenticate the client session. If the event handler does not authenticate the client, the server will perform its default authentication. To reject the credentials provided by the client, use the **SendErrorResponse** method with a result code of 401.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[OnCommand](#), [OnConnect](#)

CHttpServer::OnCommand Method

```
virtual void OnCommand(  
    UINT nClientId,  
    LPCTSTR lpszCommand,  
    LPCTSTR lpszResource,  
    LPCTSTR lpszParameters,  
    BOOL& bHandled  
);
```

A virtual method that is invoked after the client has sent a command to the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCommand

A pointer to a string that specifies the command issued by the client. The command name will always be capitalized. For a complete list of commands supported by the server, see [Server Commands](#).

lpszResource

A pointer to a string that specifies the resource that the client has requested. Depending on the command issued, it may be a document, a folder or an executable script.

lpszParameters

A pointer to a string that specifies any query parameters that have been provided by the client. The string will be empty if there were no query parameters included with the request. The query parameters in this string will be URL encoded.

bHandled

An integer value that indicates whether the handler has processed the event. If the handler sets this parameter to a non-zero value, it indicates that the application has processed the command itself. If this parameter has a value of zero, then the server will perform the default action for the command. By default, this parameter has a value of zero.

Return Value

None.

Remarks

The **OnCommand** event handler is invoked after the client has sent a command to the server, but before the command has been processed. To implement an event handler, the application should create a class derived from the **CHttpServer** class, and then override this method.

This event handler is invoked for all commands issued by the client, including invalid or disabled commands. If the **bHandled** parameter is FALSE, the server will perform the default processing for the command. If the **bHandled** parameter is set to TRUE, the server will take no action. Note that if the event handler processes the command, it must call the **SendResponse** method to send a success or error response back to the client. If this is not done, the server will consider the command to be invalid and will send a 501 "not implemented" error by default.

It is not necessary to use this event handler to disable a command. The **EnableCommand** method can be used to enable or disable specific commands, and the **IsCommandEnabled** method can be used to determine if a command is enabled.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisconnectClient](#), [EnableCommand](#), [IsCommandEnabled](#), [OnDisconnect](#), [SendResponse](#)

CHttpServer::OnConnect Method

```
virtual void OnConnect(  
    UINT nClientId,  
    LPCTSTR lpszAddress  
);
```

A virtual method that is invoked after the client has connected to the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszAddress

A pointer to a string that specifies the IP address of the client. This address may either be in IPv4 or IPv6 format, depending on how the server was configured and the address the client used to establish the connection.

Return Value

None.

Remarks

The **OnConnect** event handler is invoked after the client has connected to the server. To implement an event handler, the application should create a class derived from the **CHttpServer** class, and then override this method.

This event only occurs after the server has checked the active client limits and has accepted the connection. If the server was started with security enabled, the TLS handshake has been performed. If the server has been suspended, or the limit on the maximum number of client sessions has been exceeded, the server will terminate the client session prior to this event handler being invoked.

If this event handler is not implemented, the server will perform the default action of accepting the connection and waiting for the client to send a request for a document. To reject a connection, call the **DisconnectClient** method to terminate the client session.

Your server application should never make the assumption that for each **OnConnect** event there will be one command issued by the client. A client may issue multiple commands per session, and in some cases a client may send no commands to the server. If the client is using HTTP 1.0, connections will not be persistent. This means that the client will connect to the server, issue one command and then disconnect. For clients that use HTTP 1.1, multiple commands may be issued using a single connection. There is no guarantee that either the client or server will maintain a persistent connection, and either may request that the connection be closed after a command has completed.

If the client is a web browser, it is not unusual to see multiple, simultaneous connections being established with your server for a single request. Do not be concerned if you see multiple connections without any commands being issued. Browsers will do this to in anticipation of downloading additional assets (stylesheets, images, etc.) to improve performance and will typically close any unused connections after a few seconds. You may also see multiple requests for the file **/favicon.ico**. The browser will request this icon to display in the browser address bar and bookmarks. If the file does not exist, the browser ignores the error response from the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisconnectClient](#), [OnCommand](#), [OnDisconnect](#),

CHttpServer::OnDisconnect Method

```
virtual void OnDisconnect(  
    UINT nClientId  
);
```

A virtual method that is invoked immediately before the client is disconnected from the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

None.

Remarks

The **OnDisconnect** event handler is invoked immediately before the client is disconnected from the server. To implement an event handler, the application should create a class derived from the **CHttpServer** class, and then override this method.

This is an advisory event and it is not required for the application to explicitly disconnect the client or perform any action. The event handler cannot prevent the client from disconnecting.

Applications that implement a handler for this event should only use it to update any private data that was associated with the client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[OnConnect](#)

CHttpServer::OnDownload Method

```
virtual void OnDownload(  
    UINT nClientId,  
    LPCTSTR lpszFileName,  
    DWORD dwTimeElapsed,  
    ULARGE_INTEGER uiBytesCopied  
);
```

A virtual method that is invoked after the client has successfully downloaded a file from the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A pointer to a string that specifies the local path name of the file that was downloaded from the server. The path will always include the disk volume or share name, and the path delimiter will always be the backslash character.

dwTimeElapsed

An unsigned integer value that specifies the number of milliseconds that it took to complete the file transfer.

uiBytesCopied

An unsigned 64-bit integer value that specifies the number of bytes of data that was downloaded by the client.

Return Value

None.

Remarks

The **OnDownload** event handler is invoked after the client has successfully downloaded a file from the server using the GET command. To implement an event handler, the application should create a class derived from the **CHttpServer** class, and then override this method.

The ULARGE_INTEGER structure is actually a union that is used to represent a 64-bit value. If the compiler has built-in support for 64-bit integers, use the **QuadPart** member to access the 64-bit integer value. Otherwise, use the **LowPart** and **HighPart** members to access the value.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[OnCommand](#), [OnUpload](#)

CHttpServer::OnError Method

```
virtual void OnConnect(  
    UINT nClientId,  
    UINT nEventId,  
    DWORD dwError  
);
```

A virtual method that is invoked when the event handler encountered an error when processing a client event.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nEventId

An unsigned integer which identifies the client event that was being processed when the error occurred. For a list of event identifiers, see [Server Event Constants](#).

dwError

An unsigned integer value that specifies the error code.

Return Value

None.

Remarks

The **OnError** event handler is invoked whenever an error occurs while an event is being processed by the server. To implement an event handler, the application should create a class derived from the **CHttpServer** class, and then override this method.

It is important to note that this event is not raised for every error that occurs. The event only occurs when another event is being processed and an unhandled error occurs that must be reported back to the server application. The following are some common situations in which this event handler may be invoked:

- A network error occurs when the client connection is being accepted by the server. This could be the result of an aborted connection or some other lower-level failure reported by the networking subsystem on the server.
- The server is configured to use implicit SSL but cannot obtain the security credentials required to create the security context for the session. Usually this indicates that the server certificate cannot be found, or the certificate does not have a private key associated with it. It could also indicate a general problem with the cryptographic subsystem where the client and server could not successfully negotiate a cipher suite.
- A network error occurs when attempting to process a command issued by the client. This usually indicates that the connection to the client has been aborted, either because the client is not acknowledging the data that has been exchanged with the server, or the client has terminated abnormally. This event will not occur if the client terminates the connection normally.

In most situations where this event handler is invoked, the error is not recoverable and the only action that can be taken is to terminate the client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[GetLastError](#), [OnConnect](#), [OnCommand](#)

CHttpServer::OnExecute Method

```
virtual void OnExecute(  
    UINT nClientId,  
    LPCTSTR lpszCommand,  
    LPCTSTR lpszResource,  
    LPCTSTR lpszParameters,  
    DWORD dwExitCode  
);
```

A virtual method that is invoked after the client has successfully executed a CGI program or script on the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCommand

A pointer to a string that specifies the command issued by the client. The command name will always be capitalized. For a complete list of commands supported by the server, see [Server Commands](#).

lpszResource

A pointer to a string that specifies the resource that the client has requested.

lpszParameters

A pointer to a string that specifies any query parameters that have been provided by the client. The string will be empty if there were no query parameters included with the request. The query parameters in this string will be URL encoded.

dwExitCode

An unsigned integer that specifies the exit code that was returned by the program.

Return Value

None.

Remarks

The **OnExecute** event handler is invoked after the client has successfully executed an external CGI program or script. To implement an event handler, the application should create a class derived from the **CHttpServer** class, and then override this method.

External programs must be registered by the server application using the **RegisterProgram** method. To enable the use of scripts, the **RegisterHandler** method can be used to associate an executable program with a specific file extension.

The **GetProgramOutput** method can be used to obtain the unfiltered output from the external command, while the **GetProgramText** method will return filtered output from the program that contains only printable text characters.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetProgramOutput](#), [GetProgramText](#), [RegisterHandler](#), [RegisterProgram](#)

CHttpServer::OnResult Method

```
virtual void OnResult(  
    UINT nClientId,  
    LPCTSTR lpszCommand,  
    LPCTSTR lpszResource,  
    LPCTSTR lpszParameters,  
    UINT nResultCode  
);
```

A virtual method that is invoked after the server has processed a command issued by the client.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCommand

A pointer to a string that specifies the command issued by the client. The command name will always be capitalized. For a complete list of commands supported by the server, see [Server Commands](#).

lpszResource

A pointer to a string that specifies the resource that the client has requested. Depending on the command issued, it may be a document, a folder or an executable script.

lpszParameters

A pointer to a string that specifies any query parameters that have been provided by the client. The string will be empty if there were no query parameters included with the request. The query parameters in this string will be URL encoded.

nResultCode

A integer value that specifies the result code that was sent to the client.

Return Value

None.

Remarks

The **OnResult** event handler is invoked after the server has processed a command issued by the client. To implement an event handler, the application should create a class derived from the **CHttpServer** class, and then override this method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetCommandResult](#), [OnCommand](#)

CHttpServer::OnTimeout Method

```
virtual void OnTimeout(  
    UINT nClientId,  
    UINT nIdleTime,  
    UINT nElapsed  
);
```

A virtual method that is invoked after the client has exceeded the maximum allowed idle time.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

nIdleTime

An unsigned integer value that specifies the current idle timeout period for the client in seconds.

nElapsed

An unsigned integer value that specifies the number of seconds that the client has been idle.

Return Value

None.

Remarks

The **OnTimeout** event handler is invoked after the client has exceeded the maximum allowed idle time. To implement an event handler, the application should create a class derived from the **CHttpServer** class, and then override this method.

This event handler will be invoked prior to the client being disconnected from the server. This event will never occur while the server is returning data to the client. The **SetClientIdleTime** method can be used to change or refresh the idle timeout period for the session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[GetClientIdleTime](#), [OnCommand](#), [OnResult](#), [SetClientIdleTime](#)

CHttpServer::OnUpload Method

```
virtual void OnUpload(  
    UINT nClientId,  
    LPCTSTR lpszFileName,  
    DWORD dwTimeElapsed,  
    ULARGE_INTEGER uiBytesCopied  
);
```

A virtual method that is invoked after the client has successfully uploaded a file to the server.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A pointer to a string that specifies the local path name of the file that was uploaded to the server. The path will always include the disk volume or share name, and the path delimiter will always be the backslash character.

dwTimeElapsed

An unsigned integer value that specifies the number of milliseconds that it took to complete the file transfer.

uiBytesCopied

An unsigned 64-bit integer value that specifies the number of bytes of data that was uploaded by the client.

Return Value

None.

Remarks

The **OnUpload** event handler is invoked after the client has successfully uploaded a file to the server using the PUT command. To implement an event handler, the application should create a class derived from the **CHttpServer** class, and then override this method.

The ULARGE_INTEGER structure is actually a union that is used to represent a 64-bit value. If the compiler has built-in support for 64-bit integers, use the **QuadPart** member to access the 64-bit integer value. Otherwise, use the **LowPart** and **HighPart** members to access the value.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[OnCommand](#), [OnDownload](#)

CHttpRequest Data Structures

- HTTPCLIENTCREDENTIALS
- HTTPREQUEST
- HTTPRESPONSE
- HTTPSERVERTRANSFER

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

HTTPCLIENTCREDENTIALS Structure

The **HTTPCLIENTCREDENTIALS** structure defines the credentials used to authenticate a specific user.

```
typedef struct _HTTPCLIENTCREDENTIALS
{
    DWORD dwSize;
    DWORD dwFlags;
    UINT nAuthType;
    UINT nHostPort;
    TCHAR szHostName[HTTP_MAXHOSTNAME];
    TCHAR szUserName[HTTP_MAXUSERNAME];
    TCHAR szPassword[HTTP_MAXPASSWORD];
} HTTPCLIENTCREDENTIALS, *LPHTTPCLIENTCREDENTIALS;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwFlags

An unsigned integer value is reserved for future use. This value will always be zero.

nAuthType

An unsigned integer value that identifies the authentication method.

nHostPort

The server port number.

szHostName

A pointer to a string that specifies the server host name.

szUserName

A pointer to a string that specifies the user name.

szPassword

A pointer to a string that specifies the user password.

Remarks

When an instance of this structure is passed to the **GetClientCredentials** method, this member must be initialized to the size of the structure and all other members must be initialized with a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientCredentials](#)

HTTPREQUEST Structure

The **HTTPREQUEST** structure provides information about the request made by the client.

```
typedef struct _HTTPREQUEST
{
    DWORD    dwSize;
    DWORD    dwFlags;
    DWORD    dwVersion;
    DWORD    dwAccess;
    DWORD    dwLength;
    DWORD    dwReserved;
    UINT     nHostId;
    UINT     nHostPort;
    LPCSTR   lpszCommand;
    LPCSTR   lpszHostName;
    LPCSTR   lpszUserName;
    LPCSTR   lpszPassword;
    LPCSTR   lpszResource;
    LPCSTR   lpszParameters;
    LPCSTR   lpszDirectory;
    LPCSTR   lpszPathInfo;
    LPCSTR   lpszProgram;
    LPCSTR   lpszScriptFile;
    LPCSTR   lpszLocalFile;
    LPCSTR   lpszMediaType;
} HTTPREQUEST, *LPHTTPREQUEST;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwFlags

An unsigned integer value that provides additional information about the request. It may be one or more of the following values:

Constant	Description
HTTP_REQUEST_FLAG_PROTECTED (0x1)	The resource that the client has requested is protected. The client should only be permitted to access the resource if the client session has been authenticated.
HTTP_REQUEST_FLAG_PROGRAM (0x2)	The resource that the client has requested is a CGI program or an executable script. The output from the program or script should be returned to the client in the response. If this flag is specified, the <i>lpszProgram</i> member of the structure specifies the name of the program that should be executed.
HTTP_REQUEST_FLAG_SCRIPT (0x4)	The resource that the client has requested is an executable script. This flag is set when the request URL is mapped to a registered script handler. The <i>lpszProgram</i> member specifies the name of the program that is responsible for executing the

script, and the *lpszScriptFile* parameter specifies the full path to the script itself.

dwVersion

An unsigned integer value that specifies the protocol version used. It may be one of the following values:

Constant	Description
HTTP_VERSION_09 (0x00009)	The client issued a GET command without specifying a protocol version.
HTTP_VERSION_10 (0x10000)	The client issued a command that specified version 1.0 of the protocol.
HTTP_VERSION_11 (0x10001)	The client issued a command that specified version 1.1 of the protocol.

dwAccess

An unsigned integer value that specifies the access permissions that were assigned to the resource. For a list of file access permissions, see [User and File Access Constants](#).

dwLength

An unsigned integer value that will specify the number of bytes of request data provided by the client. If the client did not submit any data with the request, this member will have a value of zero.

dwReserved

An unsigned integer value that is reserved for future use.

nHostId

An integer value that identifies the virtual host that was specified by the client. This is based on the value of the Host request header included in the request. This value will be zero if a host name is not specified by the client, or does not match one of the virtual hosts that have been created.

nHostPort

An integer value that specifies the port number that the client used to establish the connection. This will be the same port number that was used when starting the server.

lpszCommand

A pointer to a string that specifies the command that was issued by the client. The command will always be in upper case. Refer to [Hypertext Transfer Protocol Commands](#) for a list of standard commands.

lpszHostName

A pointer to a string that identifies the hostname or IP address that the client used to establish the connection. This will typically correspond to the hostname assigned to the server, or to one of the virtual hosts that have been created.

lpszUserName

A pointer to a string that specifies the username provided with the request. This member is only set if the client has included authentication credentials as part of the request. If the client has not provided any credentials, this member will be NULL.

lpszPassword

A pointer to a string that specifies the password provided with the request. This member is only set if the client has included authentication credentials as part of the request. If the client has not provided any credentials, this member will be NULL.

lpszResource

A pointer to a string that specifies the URL path provided by the client.

lpszParameters

A pointer to a string that specifies any query parameters that were included in the request made by the client. If the URL did not include any query parameters, this member will be NULL.

lpszDirectory

A pointer to a string that specifies the full path to the root directory for the virtual host.

lpszPathInfo

A pointer to a string that specifies additional path information provided by the client when referencing an executable CGI program or script in the URL. For example, if a program is mapped to the virtual path "/orders/invoice" and the client requests "/orders/invoice/pdf/10001" this member will be the string "/pdf/10001". If there is no path information associated with the request, this member will be NULL.

lpszProgram

A pointer to a string that specifies the local path to the CGI program that should be executed to process the request made by the client. If this member is NULL, it indicates that the client has provided a URL that maps to a local file or directory.

lpszScriptFile

A pointer to a string that specifies the local path to the script file that will be executed by the handler. If this member is NULL, it indicates that the request URL was not mapped to registered script handler.

lpszLocalFile

A pointer to a string that specifies the local path to the directory or file name referenced by the URL. If the *lpszPathInfo* member is not NULL, then this member will point to a local file name based on the path information appended to the root directory for the virtual host.

lpszMediaType

A pointer to a string that identifies the request data using the standard Internet media types defined in RFC 2046. It is used to designate the format for various types of content and has two parts, a primary and secondary media type separated by a forward slash. Common examples are "text/plain", "text/html" and "application/octet-stream". If the client did not submit any data with the request, this member will be NULL.

Remarks

This structure is used with the **ReceiveRequest** method and all members should be initialized to a value of zero, except for the *dwSize* member which should be initialized to size of the structure. Failure to properly initialize the structure will cause the method call to fail.

The value of the *lpszLocalFile* member depends on whether the client has requested a static document, or if the URL is mapped to a registered program or script handler. If the request is for a static document, then the *dwFlags* member will not have the HTTP_REQUEST_FLAG_PROGRAM bit flag set and the *lpszLocalFile* member will be the full path to the document. If the URL is mapped to a registered program or script, then the *lpszLocalFile* member will be the full path to the server root directory. If the request URL included additional path information, that will be appended to the root directory.

The value of the *lpzScriptFile* member is only defined if the *dwFlags* member has the HTTP_REQUEST_FLAG_SCRIPT bit flag set. This indicates that the request URL references a file that is an executable script with a handler that was registered using the **RegisterHandler** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ReceiveRequest](#), [SendResponse](#), [HTTPRESPONSE](#)

HTTPRESPONSE Structure

The **HTTPRESPONSE** structure provides additional information about the response to the client.

```
typedef struct _HTTPRESPONSE
{
    DWORD    dwSize;
    DWORD    dwFlags;
    DWORD    dwVersion;
    DWORD    dwReserved;
    UINT     nExpires;
    UINT     nResultCode;
    LPCTSTR  lpszReason;
    LPCTSTR  lpszMediaType;
} HTTPRESPONSE, *LPHTTPRESPONSE;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwFlags

An unsigned integer value that specifies one or more response flags. This member is reserved for future use and should always have a value of zero.

dwVersion

An unsigned integer value that specifies the protocol version used. It may be one of the following values:

Constant	Description
HTTP_VERSION_DEFAULT (0)	The server should respond to the client using the same version specified in the request.
HTTP_VERISON_10 (0x10000)	The server should respond to the client using version 1.0 of the protocol.
HTTP_VERSION_11 (0x10001)	The server should respond to the client using version 1.1 of the protocol.

dwReserved

An unsigned integer value that is reserved for future use.

nExpires

An integer value that specifies the number of seconds until the client should consider any cached response data to be stale. If this value is zero, the default cache expiration time for the server will be used. The default cache expiration time is 7200 seconds (2 hours). The value of this structure member is ignored if the HTTP_RESPONSE_NOCACHE option is specified when the response is sent to the client.

nResultCode

An integer value that specifies the result code that should be sent in response to the client request. The result code is a three-digit number that indicates success or failure. For more information, refer to the **GetCommandResult** method.

lpszReason

A string that describes the result code sent to the client. The description should be brief

and should not contain any formatting characters or HTML markup. This parameter may be NULL, in which case a default description of the result code will be used.

lpzMediaType

A string that specifies the Internet media type for the data that is being sent to the client as defined in RFC 2046. The format for the content type string consists of two parts, a primary and secondary media type separated by a forward slash. Common examples are "text/plain", "text/html" and "application/octet-stream". If this member is NULL, the library will attempt to automatically determine the appropriate media type.

Remarks

This structure is used with the **SendResponse** method and the *dwSize* member must be initialized to size of the structure. Failure to properly initialize the structure will cause the method call to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetCommandResult](#), [ReceiveRequest](#), [SendResponse](#), [HTTPREQUEST](#)

HTTPSERVERTRANSFER Structure

The **HTTPSERVERTRANSFER** structure provides information about the last file transfer performed by a client.

```
typedef struct _HTTPSERVERTRANSFER
{
    DWORD          dwSize;
    DWORD          dwReserved;
    DWORD          dwFileAccess;
    DWORD          dwTimeElapsed;
    ULARGE_INTEGER uiBytesCopied;
    TCHAR          szFileName[MAX_PATH];
} HTTPSERVERTRANSFER, *LPHTTPSERVERTRANSFER;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwReserved

An unsigned integer value that is reserved for future use. This value will always be zero.

dwFileAccess

An unsigned integer value that specifies the how the local file was accessed. It can be one of the following values:

Constant	Description
HTTP_FILE_READ (0)	The file was opened for reading. This mode indicates that the client issued the GET command to download the contents of a file from the server to the client system. The <i>szFileName</i> member specifies the name of the local file on the server that was downloaded by the client.
HTTP_FILE_WRITE (1)	The file was opened for writing. This mode indicates that the client issued the PUT command to upload the contents of a file from the client system to server. The <i>szFileName</i> member specifies the name of the local file on the server that was created by the client. If a file already existed with the name name, it was replaced.

dwTimeElapsed

The amount of time that it took for the file transfer to complete in milliseconds. This value is limited to the resolution of the system timer, which is typically in the range of 10 to 16 milliseconds. This value may be zero if the transfer occurred over a local network or on the same host using a loopback address.

uiBytesCopied

A 64-bit integer value that specifies the total number of bytes copied during the file transfer. This value is represented by a ULARGE_INTEGER union which provides support for those programming languages that do not have intrinsic support for 64-bit integers. For more information, refer to the Windows SDK documentation. The application should not make the assumption that this is the actual size of the file.

szFileName

A pointer to a string value that will contain the full path to the local file that was transferred. The

dwFileAccess member determines whether the file name represents a file that was downloaded by the client, or uploaded from the client and stored on the server.

Remarks

When an instance of this structure is passed to the **GetTransferInfo** method, the *dwSize* member must be initialized to the size of the structure, otherwise the method will fail with an error indicating that the parameter is invalid. All other members should be initialized to a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetTransferInfo](#)

SECURITYCREDENTIALS Structure

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on

	what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that

will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A pointer to a null terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A pointer to a null terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A pointer to a null terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A pointer to a null terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
------------	-------------

CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
----	---

MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
----	--

ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.
------	--

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

A pointer to a null terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used for SSH connections

and should always be NULL when establishing a secure connection using SSL or TLS.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* member should be set to CREDENTIAL_STORE_FILENAME and the *lpzCertStore* member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS#12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the *lpzPassword* member or the library will be unable to access the certificate.

Note that the *lpzUserName* and *lpzPassword* members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

The TLS 1.1 and TLS 1.2 protocols are only supported on Windows 7, Windows Server 2008 R2 and later versions of the platform. If these options are specified and the application is running on Windows XP or Windows Vista, the protocol version will be downgraded to TLS 1.0 for backwards compatibility with those platforms.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

Internet Control Message Protocol Class Library

Determine if a remote host is reachable and how packets of data are routed to that system.

Reference

- [Class Methods](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

Class Name	CIcmpClient
File Name	CSICMV10.DLL
Version	10.0.1468.2518
LibID	A8CDC210-0CEC-4A3B-A367-A649CEEA7419
Import Library	CSICMV10.LIB
Dependencies	None
Standards	RFC 792

Overview

The Internet Control Message Protocol (ICMP) is commonly used to determine if a remote host is reachable and how packets of data are routed to that system. Users are most familiar with this protocol as it is implemented in the ping and traceroute command line utilities. The ping command is used to check if a system is reachable and the amount of time that it takes for a packet of data to make a round trip from the local system, to the remote host and then back again. The traceroute command is used to trace the route that a packet of data takes from the local system to the remote host, and can be used to identify potential problems with overall throughput and latency. The library can be used to build in this type of functionality in your own applications, giving you the ability to send and receive ICMP echo datagrams in order to perform your own analysis.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the

file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Internet Control Message Protocol Class Methods

Class	Description
CicmpClient	Constructor which initializes the current instance of the class
~CicmpClient	Destructor which releases resources allocated by the class
Method	Description
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified client handle to another thread
Cancel	Cancel the current blocking operation
DetachHandle	Detach the handle for the current instance of this class
DisableEvents	Disable asynchronous event notification
DisableTrace	Disable logging of socket-level calls to the trace log
Echo	Send one or more echo datagrams to the specified host
EnableEvents	Enable asynchronous event notification
EnableTrace	Enable logging of socket function calls to a file
FormatAddress	Convert a numeric IP address to a string
FreezeEvents	Suspend or resume event handling by the calling process
GetErrorString	Return a description for the specified error code
GetFirstHost	Return the first host from the traceroute
GetHandle	Return the client handle used by this instance of the class
GetHostAddress	Return the current host IP address
GetHostName	Return the current host name
GetLastError	Return the last error code
GetNextHost	Return the next host from the traceroute
GetPacketSize	Return the ICMP datagram packet size
GetRecvCount	Return the number of packets received
GetSendCount	Return the number of packets sent
GetSequenceId	Return the current sequence identifier
GetTimeout	Return the number of milliseconds until an operation times out
GetTimeToLive	Return the current time-to-live for the ICMP datagram
GetTripTime	Return the current trip time statistics
IcmpEventProc	Callback method that processes events generated by the client
IsBlocking	Determine if the client is blocked, waiting for information
IsInitialized	Determine if the class has been successfully initialized

RecvEcho	Read an ICMP datagram returned by the remote host
RegisterEvent	Register an event callback function
Reset	Reset the current client state
ResolveAddress	Resolve an IP address into a fully qualified host name
SendEcho	Send an ICMP datagram to the specified host
SetHostAddress	Set the IP address of the host to receive the next datagram
SetHostName	Set the name of the host to receive the next datagram
SetLastError	Set the last error code
SetPacketSize	Set the ICMP datagram packet size
SetSequenceId	Set the sequence identifier for the next datagram
SetTimeout	Set the number of milliseconds until an operation times out
SetTimeToLive	Set the time-to-live for the next datagram
ShowError	Display a message box with a description of the specified error
TraceRoute	Trace the route from the local system to a remote host

CIcmpClient::CIcmpClient Method

CIcmpClient();

The **CIcmpClient** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **IcmpInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[~CIcmpClient](#), [IsInitialized](#)

CicmpClient::~CicmpClient

`~CicmpClient();`

The **CicmpClient** destructor releases resources allocated by the current instance of the **CicmpClient** object. It also uninitialized the library if there are no other concurrent uses of the class.

Remarks

When a **CicmpClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CicmpClient](#)

CIcmpClient::AttachHandle Method

```
VOID AttachHandle(  
    HCLIENT hClient  
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle already has been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the *hClient* parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[AttachThread](#), [DetachHandle](#), [GetHandle](#)

CIcmpClient::AttachThread Method

```
DWORD AttachThread(  
    DWORD dwThreadId  
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[AttachHandle](#), [Cancel](#), [CloseHandle](#), [CreateHandle](#), [DetachHandle](#), [GetHandle](#)

CIcmpClient::Cancel Method

```
INT Cancel();
```

The **Cancel** method cancels any outstanding blocking operation in the client, causing the blocking method to fail. The application may then retry the operation or terminate the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[IsBlocking](#)

CIcmpClient::CloseHandle Method

```
INT CloseHandle(  
    HCLIENT hClient  
);
```

The **CloseHandle** method closes the socket and releases the memory allocated for the client session.

Parameters

hClient

A handle to the client session.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CIcmpClient](#), [CreateHandle](#)

ICmpClient::CreateHandle Method

```
HCLIENT CreateHandle(  
    UINT nPacketSize,  
    UINT nTimeToLive,  
    DWORD dwTimeout,  
    DWORD dwReserved,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **CreateHandle** method creates a client handle for sending and receiving ICMP echo datagrams. If an event notification window is specified, the client will be notified when a network event occurs.

Parameters

nPacketSize

An unsigned integer which specifies the default packet size used when generating ICMP echo datagrams. The minimum packet size is 32 bytes and the maximum size is 65,535 bytes.

nTimeToLive

An unsigned integer which specifies the default time-to-live for ICMP echo datagrams. This determines the maximum number of times that a packet will be routed from one system to another while enroute to its destination. The minimum time-to-live value is 1, the maximum is 255. The recommend value for this parameter is 255, and typical applications should use a time-to-live value of at least 30.

dwTimeout

An unsigned integer which specifies the maximum number of milliseconds to wait before the current operation times out.

dwReserved

A reserved parameter. This value should always be zero.

hEventWnd

The handle to an asynchronous notification window. This window receives messages which notify the client when asynchronous network events occur. If asynchronous event notification is not required, this parameter may be NULL.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the *hEventWnd* parameter is NULL, this parameter should be specified as WM_NULL.

Return Value

If the method succeeds, the return value is a handle to the client session. If the method fails, the return value is INVALID_CLIENT. To get extended error information, call **GetLastError**.

Remarks

The **CreateHandle** method creates a client handle that is used with subsequent calls to the library. This library uses a special type of socket called a raw socket, which is created to send and receive ICMP echo datagrams. Raw socket support is optional under the Windows Sockets specification, and may not be available if a non-standard networking libraries are used or may only be available

to privileged accounts.

If the *hEventWnd* parameter is not NULL, the client operates in asynchronous mode and messages will be posted to the notification window when a network event occurs. When a message is posted to the window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
ICMP_EVENT_ECHO	The client has generated an ICMP echo datagram and has sent it to the specified host. If the datagram is received, the remote host should generate a reply and return it to the sender.
ICMP_EVENT_REPLY	The client has received an ICMP echo reply datagram from the remote host. At this point the client can collect statistical information.
ICMP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation.
ICMP_EVENT_CANCEL	The current operation has been canceled. The client application may attempt to retry the operation or close the handle.

To cancel asynchronous notification and return the client to a blocking mode, use the **DisableEvents** method.

The ability to create and send ICMP echo datagrams is limited to privileged users. Non-administrator users will receive an error if they attempt to create a client handle. On Windows NT it is possible to disable this security check by creating or modifying the system registry. Microsoft Knowledge Base article 195445 has additional information and instructions for making this change.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Sockets 2.0 update or later.

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[CicmpClient](#), [CloseHandle](#)

CIcmpClient::DetachHandle Method

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csicmv10.lib`

See Also

[AttachHandle](#), [GetHandle](#)

CIcmpClient::DisableEvents Method

```
INT DisableEvents();
```

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the client has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[EnableEvents](#), [RegisterEvent](#)

CIcmpClient::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[EnableTrace](#)

ICmpClient::Echo Method

```
INT Echo(  
    LPCTSTR lpszHostName,  
    UINT nIterations,  
    UINT nPacketSize,  
    DWORD dwTimeout,  
    LPICMPTIME lpTime  
);  
  
INT Echo(  
    LPCTSTR lpszHostName,  
    LPICMPTIME lpTime  
);
```

The **Echo** method sends one or more ICMP echo datagrams, collecting information about the reliability and latency of a connection between the local and remote host.

Parameters

lpszHostName

A pointer to a string which specifies the fully qualified domain name of the remote host, or the IP address in dotted notation.

nIterations

An unsigned integer value which specifies the number of echo datagrams that will be sent to the remote host. The minimum value for this parameter is 1 and the maximum value is 512.

nPacketSize

An unsigned integer value which specifies the size of the echo datagram in bytes. The minimum size is 1 byte and the maximum size is 65,535 bytes. It is recommended that most applications use the minimum size of 32 bytes for this parameter.

dwTimeout

An unsigned integer which specifies the number of milliseconds the method will wait for a response to an echo datagram.

lpTime

A pointer to an [ICMPTIME](#) structure which will contain the minimum, maximum and average round trip times for the echo datagrams sent and received.

Return Value

If the method succeeds, the return value is the number of replies received from the remote host. If the method fails, the return value is `ICMP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **Echo** method sends a series of ICMP echo datagrams to the specified host, providing a simplified interface for pinging a remote system. If the method returns the same value as the number of iterations, then replies were received for all of the echo datagrams that were sent. This would typically indicate that the client can establish a reliable connection to the server; the values returned in the `ICMPTIME` structure provide information on the latency between the two hosts. Higher average time values would indicate greater latency and reduced throughput between the systems. If the method returns a value less than the specified number of iterations, this indicates that replies were not received for one or more of the echo datagrams. This may indicate that there

are problems routing data between the local and remote host. A return value of zero indicates that there was no response to the echo datagrams. The remote host may not exist or may not be available.

The failure for a host to respond to an ICMP echo datagram may not indicate a problem with the remote system. In some cases, a router between the local and remote host may be malfunctioning or discarding the datagrams. Systems can also be configured to specifically ignore ICMP echo datagrams and not respond to them; this is often a security measure to prevent certain kinds of Denial of Service attacks.

The ability to send ICMP datagrams may be restricted to users with administrative privileges, depending on the policies and configuration of the local system. If you are unable to send or receive any ICMP datagrams, it is recommended that you check the firewall settings and any third-party security software that could impact the normal operation of this component.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[CreateHandle](#), [GetTripTime](#), [RecvEcho](#), [SendEcho](#), [TraceRoute](#)

ICmpClient::EnableEvents Method

```
INT EnableEvents(  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and is retained for backwards compatibility. Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Constant	Description
ICMP_EVENT_ECHO	The client has generated an ICMP echo datagram and has sent it to the specified host. If the datagram is received, the remote host should generate a reply and return it to the sender.
ICMP_EVENT_REPLY	The client has received an ICMP echo reply datagram from the remote host. At this point the client can collect statistical information.
ICMP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
ICMP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csicmv10.lib

See Also

[DisableEvents](#), [RegisterEvent](#)

CIcmpClient::EnableTrace Method

```
BOOL EnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **EnableTrace** method enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the remote host.

Trace method logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableTrace](#)

CIcmpClient::FormatAddress Method

```
INT FormatAddress(  
    DWORD dwAddress,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

```
INT FormatAddress(  
    DWORD dwAddress,  
    CString& strAddress  
);
```

The **FormatAddress** method converts a numeric IP address to a null-terminated string.

Parameters

dwAddress

A unsigned 32-bit integer which specifies the IP address to be converted into a string.

lpszAddress

A pointer to a null-terminated array of characters which will contain the converted IP address in dot-notation. This string should be at least 16 characters in length. An alternate form of this method accepts a **CString** object which will contain the IP address in dotted notation when the method returns.

nMaxLength

The maximum number of characters which may be copied in to the string buffer.

Return Value

If the method succeeds, the return value is the length of the string buffer. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostAddress](#), [GetHostName](#), [ResolveAddress](#)

CIcmpClient::FreezeEvents Method

```
INT FreezeEvents(  
    BOOL bFreeze  
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the client.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableEvents](#), [EnableEvents](#), [RegisterEvent](#)

CIcmpClient::GetErrorString Method

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);  
  
INT GetErrorString(  
    DWORD dwErrorCode,  
    CString& strDescription  
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLastError](#), [SetLastError](#), [ShowError](#)

ICmpClient::GetFirstHost Method

```
BOOL GetFirstHost(  
    ICMPTRACE& icmpTrace  
);
```

The **GetFirstHost** method returns information about the first host between the local system and the remote host specified in the call to the **TraceRoute** method.

Parameters

icmpTrace

An [ICMPTRACE](#) structure which contains information about the intermediate host.

Return Value

If the method succeeds, the return value is non-zero. If the return value is zero, this indicates that no host information is available. To get extended error information, call **GetLastError**.

Remarks

The **GetFirstHost** method is used in conjunction with the **GetNextHost** method to enumerate all of the intermediate hosts between the local system and the remote host specified when the **TraceRoute** method was called.

Example

```
BOOL bResult = pClient->TraceRoute(strAddress);  
if (bResult)  
{  
    ICMPTRACE icmpTrace;  
  
    bResult = pClient->GetFirstHost(icmpTrace);  
    while (bResult)  
    {  
        // The icmpTrace structure contains information about the  
        // intermediate host in the traceroute  
        CString strHostName;  
        pClient->ResolveAddress(icmpTrace.dwHostAddress, strHostName);  
  
        bResult = pClient->GetNextHost(icmpTrace);  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetNextHost](#), [ResolveAddress](#), [TraceRoute](#)

CIcmpClient::GetHandle Method

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csicmv10.lib`

See Also

[AttachHandle](#), [DetachHandle](#), [IsInitialized](#)

CIcmpClient::GetHostAddress Method

```
INT GetHostAddress(  
    LPTSTR lpszHostAddress,  
    INT cbHostAddress  
);  
  
INT GetHostAddress(  
    CString& strHostAddress  
);
```

The **GetHostAddress** copies the IP address of the current host into the specified buffer as a string using dot-notation.

Parameters

lpszHostAddress

A pointer to the buffer that will contain the IP address of the current remote host in dot-notation. This buffer should be at least 16 characters in length.

cbHostAddress

The maximum number of characters that may be copied into the buffer, including the terminating null character character.

Return Value

If the method succeeds, the return value is the length of the address string. If the return value is zero, this indicates that no remote host has been specified. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FormatAddress](#), [GetHostName](#), [ResolveAddress](#), [SetHostAddress](#), [SetHostName](#)

CIcmpClient::GetHostName Method

```
INT GetHostName(  
    LPTSTR lpszHostName,  
    INT cbHostName  
);  
  
INT GetHostName(  
    CString& strHostName  
);
```

The **GetHostName** copies the name of the current host into the specified buffer.

Parameters

lpszHostName

A pointer to the buffer that will contain the name of the current remote host. This buffer should be at least 64 characters in length.

cbHostName

The maximum number of characters that may be copied into the buffer, including the terminating null character character.

Return Value

If the method succeeds, the return value is the length of the host name string. If the return value is zero, this indicates that no remote host has been specified. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FormatAddress](#), [GetHostAddress](#), [ResolveAddress](#), [SetHostAddress](#), [SetHostName](#)

ICmpClient::GetLastError Method

```
DWORD GetLastError();  
  
DWORD GetLastError(  
    CString& strDescription  
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError(0)** when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or ICMP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[GetErrorString](#), [SetLastError](#), [ShowError](#)

ICmpClient::GetNextHost Method

```
BOOL GetFirstHost(  
    ICMPTRACE& icmpTrace  
);
```

The **GetNextHost** method returns information about the next host between the local system and the remote host specified in the call to the **TraceRoute** method.

Parameters

icmpTrace

An [ICMPTRACE](#) structure which contains information about the intermediate host.

Return Value

If the method succeeds, the return value is non-zero. If the return value is zero, this indicates that no host information is available. To get extended error information, call **GetLastError**.

Remarks

The **GetNextHost** method is used in conjunction with the **GetFirstHost** method to enumerate all of the intermediate hosts between the local system and the remote host specified when the **TraceRoute** method was called.

Example

```
BOOL bResult = pClient->TraceRoute(strAddress);  
if (bResult)  
{  
    ICMPTRACE icmpTrace;  
  
    bResult = pClient->GetFirstHost(icmpTrace);  
    while (bResult)  
    {  
        // The icmpTrace structure contains information about the  
        // intermediate host in the traceroute  
        CString strHostName;  
        pClient->ResolveAddress(icmpTrace.dwHostAddress, strHostName);  
  
        bResult = pClient->GetNextHost(icmpTrace);  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFirstHost](#), [ResolveAddress](#), [TraceRoute](#)

CIcmpClient::GetPacketSize Method

```
UINT GetPacketSize();
```

The **GetPacketSize** method returns the size of the ICMP echo datagram that will be sent to the remote host. The minimum packet size is 1 byte, the maximum packet size is 65,535 bytes.

Parameters

None.

Return Value

If the method succeeds, the return value is the size of the datagram. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetRecvCount](#), [GetSendCount](#), [GetSequenceId](#), [GetTimeToLive](#), [GetTripTime](#), [SetPacketSize](#), [SetSequenceId](#), [SetTimeToLive](#)

CIcmpClient::GetRecvCount Method

```
INT GetRecvCount();
```

The **GetRecvCount** method returns the number of replies sent to the client from the remote host.

Parameters

None.

Return Value

If the method succeeds, the return value is the number of replies. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetPacketSize](#), [GetSendCount](#), [GetSequenceId](#), [GetTimeToLive](#), [GetTripTime](#), [SetSequenceId](#), [SetTimeToLive](#)

CIcmpClient::GetSendCount Method

```
INT GetSendCount();
```

The **GetSendCount** method returns the number of datagrams sent to the remote host by the client.

Parameters

None.

Return Value

If the method succeeds, the return value is the number of datagrams sent by the client. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetPacketSize](#), [GetRecvCount](#), [GetSequenceId](#), [GetTimeToLive](#), [GetTripTime](#), [SetSequenceId](#), [SetTimeToLive](#)

CIcmpClient::GetSequenceId Method

```
INT GetSequenceId();
```

The **GetSequenceId** method returns the sequence identifier for the last ICMP echo datagram received by the client.

Parameters

None.

Return Value

If the method succeeds, the return value is the sequence identifier. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetPacketSize](#), [GetRecvCount](#), [GetSendCount](#), [GetTimeToLive](#), [GetTripTime](#), [SetSequenceId](#), [SetTimeToLive](#)

CIcmpClient::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the remote host. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[CreateHandle](#), [SetTimeout](#)

CIcmpClient::GetTimeToLive Method

```
INT GetTimeToLive();
```

The **GetTimeToLive** method returns the time-to-live for the last ICMP echo datagram received by the client.

Parameters

None.

Return Value

If the method succeeds, the return value is the time-to-live for the last datagram. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The time-to-live (TTL) value is specified in the IP header of a datagram, and is used to control the number of routers that the datagram is passed through. Each router that handles the datagram decrements the TTL value by one. When it drops to zero, a datagram is returned to the sender, specifying that the TTL has been exceeded. The default value is 255.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[GetPacketSize](#), [GetRecvCount](#), [GetSendCount](#), [GetSequenceId](#), [GetTripTime](#), [SetSequenceId](#), [SetTimeToLive](#)

ICmpClient::GetTripTime Method

```
INT GetTripTime(  
    LPICMPTIME lpTime  
);
```

The **GetTripTime** returns round-trip statistics for the datagrams sent to the current remote host.

Parameters

lpTime

A pointer to an [ICMPTIME](#) data structure which will contain the round-trip statistics for the current remote host.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `ICMP_ERROR`. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csicmv10.lib`

See Also

[GetPacketSize](#), [GetRecvCount](#), [GetSendCount](#), [GetSequenceId](#), [GetTimeToLive](#), [SetSequenceId](#), [SetTimeToLive](#)

CIcmpClient::IsBlocking Method

BOOL IsBlocking();

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking method may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Cancel](#), [IsInitialized](#)

CIcmpClient::IsInitialized Method

BOOL IsInitialized();

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csicmv10.lib

See Also

[CIcmpClient](#), [IsBlocking](#)

CIcmpClient::RecvEcho Method

```
INT RecvEcho();
```

The **RecvEcho** method reads the reply to an ICMP echo datagram generated by the client. This method should only be called by asynchronous client sessions in response to an event notification that a datagram has been received.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[GetRecvCount](#), [GetSequenceId](#), [GetTimeToLive](#), [SendEcho](#)

ICmpClient::RegisterEvent Method

```
INT RegisterEvent(  
    UINT nEventId,  
    ICMPEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Constant	Description
ICMP_EVENT_ECHO	The client has generated an ICMP echo datagram and has sent it to the specified host. If the datagram is received, the remote host should generate a reply and return it to the sender.
ICMP_EVENT_REPLY	The client has received an ICMP echo reply datagram from the remote host. At this point the client can collect statistical information.
ICMP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
ICMP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **IcmpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **IcmpEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[DisableEvents](#), [EnableEvents](#), [FreezeEvents](#), [IcmpEventProc](#)

ICmpClient::Reset Method

```
INT Reset();
```

The **Reset** method resets the client session, clearing the packet trip statistics, time-to-live, sequence identifier, send and receive counts.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[SetHostAddress](#), [SetHostName](#), [SetSequenceId](#), [SetTimeToLive](#)

ICmpClient::ResolveAddress Method

```
INT ResolveAddress(  
    DWORD dwAddress,  
    LPTSTR lpszHostName,  
    INT nMaxLength  
);  
  
INT ResolveAddress(  
    DWORD dwAddress,  
    CString& strHostName  
);
```

The **ResolveAddress** method resolves a numeric IP address into a fully qualified domain name.

Parameters

dwAddress

The IP address to be resolved, specified as an unsigned 32-bit integer in network byte order.

lpszHostName

A pointer to a buffer that will contain a null-terminated string that specifies the fully qualified domain name for the host. It is recommended that this buffer be at least 64 characters in length.

nMaxLength

The maximum number of characters that can be copied into the string buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the host name. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **ResolveAddress** method is used to perform a reverse DNS lookup, converting a numeric IP address into a fully qualified domain name. This is useful for methods like **TraceRoute**, which return host addresses as numeric values in network byte order. If the reverse DNS lookup fails because there is no PTR record for the given IP address, a printable form of the address in dotted notation will be returned in the string buffer.

Calling this method will cause the thread to block until the IP address is resolved, or until the DNS query times out because there is no reverse record for the address. In some cases, a reverse lookup can take several seconds.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csicmv10.lib

See Also

[GetHostAddress](#), [GetHostName](#), [SetHostAddress](#), [SetHostName](#), [TraceRoute](#)

CIcmpClient::SendEcho Method

```
INT SendEcho();
```

The **SendEcho** method sends an ICMP echo datagram to the remote host.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

For asynchronous client sessions, this method returns immediately. Otherwise, the client enters a blocking state and waits for a reply from the remote host. The method returns when a reply has been received, or the operation times-out.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Echo](#), [GetSendCount](#), [RecvEcho](#), [SetSequenceId](#), [SetTimeToLive](#), [TraceRoute](#)

ICmpClient::SetHostAddress Method

```
INT SetHostAddress(  
    DWORD dwAddress  
);
```

```
INT SetHostAddress(  
    LPCTSTR lpszAddress  
);
```

The **SetHostAddress** method specifies the IP address of the host to receive an ICMP echo datagram. If the address specifies a new host, the current client statistics are reset.

Parameters

dwAddress

The IP address of the remote host as a 32-bit integer value, specified in network byte order. In an alternate form of this method, a pointer to a string which specifies the IP address in dotted notation may also be used.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

To specify a remote host name, use the **SetHostName** method instead.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[GetHostAddress](#), [GetHostName](#), [SetHostName](#)

ICmpClient::SetHostName Method

```
INT SetHostName(  
    HCLIENT hClient,  
    LPCTSTR lpszHostName  
);
```

The **SetHostName** method specifies the name of the host to receive an ICMP echo datagram. If the name specifies a new host, the current client statistics are reset.

Parameters

hClient

A handle to the client session.

lpszHostName

A pointer to a string which specifies the name, or IP address in dot-notation, of the remote host.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostAddress](#), [GetHostName](#), [SetHostAddress](#)

ICmpClient::SetLastError Method

```
VOID SetLastError(  
    DWORD dwErrorCode  
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or ICMP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[GetErrorString](#), [GetLastError](#), [ShowError](#)

ICmpClient::SetPacketSize Method

```
UINT SetPacketSize(  
    UINT nPacketSize  
);
```

The **SetPacketSize** method sets the size of the ICMP datagram packet that is sent to the remote host.

Parameters

nPacketSize

Size of the ICMP datagram packet in bytes. The minimum packet size is 1 byte and the maximum packet size is 65,535 bytes.

Return Value

If the method succeeds, the return value is the previous ICMP datagram packet size. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

Note that packet sizes over 512 bytes may not be supported by your networking software or configuration. It is recommended that most applications use the minimum packet size of 32 bytes.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[GetPacketSize](#), [SetSequenceId](#), [SetTimeToLive](#)

ICmpClient::SetSequenceId Method

```
INT SetSequenceId(  
    INT nSequenceId  
);
```

The **SetSequenceId** method sets the sequence identifier for the next ICMP echo datagram sent by the client. The default sequence identifier for the first datagram is one.

Parameters

nSequenceId

The sequence identifier for the next datagram sent by the client.

Return Value

If the method succeeds, the return value is the previous sequence identifier. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[GetPacketSize](#), [GetRecvCount](#), [GetSendCount](#), [GetSequenceId](#), [GetTimeToLive](#), [GetTripTime](#), [SetPacketSize](#), [SetTimeToLive](#)

CIcmpClient::SetTimeout Method

```
INT SetTimeout(  
    UINT nTimeout  
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the remote host. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: csicmv10.lib

See Also

[GetTimeout](#)

CIcmpClient::SetTimeToLive Method

```
INT SetTimeToLive(  
    INT nTimeToLive  
);
```

The **SetTimeToLive** method sets the maximum time-to-live for the next ICMP datagram sent by the client.

Parameters

nTimeToLive

The time-to-live value for the next ICMP echo datagram.

Return Value

If the method succeeds, the return value is the previous time-to-live value. If the method fails, the return value is ICMP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The time-to-live (TTL) value is specified in the IP header of a datagram, and is used to control the number of routers that the datagram is passed through. Each router that handles the datagram decrements the TTL value by one. When it drops to zero, a datagram is returned to the sender, specifying that the TTL has been exceeded.

Calling this method changes the default TTL value for all subsequent ICMP datagrams sent by the library, with the default value being 255. Note that not all Windows Sockets implementations support setting the time-to-live value.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[GetPacketSize](#), [GetRecvCount](#), [GetSendCount](#), [GetSequenceId](#), [GetTimeToLive](#), [GetTripTime](#), [SetPacketSize](#), [SetSequenceId](#)

CIcmpClient::ShowError Method

```
INT ShowError(  
    LPCTSTR lpszAppTitle,  
    UINT uType,  
    DWORD dwErrorCode  
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A pointer to a string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetErrorString](#), [GetLastError](#), [SetLastError](#)

ICmpClient::TraceRoute Method

```
BOOL TraceRoute(  
    LPCTSTR lpszHostName,  
    UINT nMaxHops,  
    DWORD dwTimeout  
);
```

The **TraceRoute** method sends a series of ICMP echo datagrams to trace the route taken from the local system to the remote host.

Parameters

lpszHostName

A pointer to a string which specifies the fully qualified domain name of the remote host, or the IP address in dotted notation.

nMaxHops

An unsigned integer which specifies the maximum number of routers the datagram will be forwarded through (the number of hops) to the remote host. The minimum value is 1 and the maximum value is 255. It is recommended that most applications specify a value of at least 30.

dwTimeout

An unsigned integer which specifies the number of milliseconds the method will wait for a response to an echo datagram.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **TraceRoute** method sends a series of ICMP echo datagrams to the specified host, adjusting the time-to-live value to determine the intermediate hosts that route the packet. Use the **GetFirstHost** and **GetNextHost** methods to enumerate the hosts returned by the method.

It is important to note that the failure of an intermediate host to respond to an ICMP echo datagram may not indicate a problem with the remote system. Systems can be configured to specifically ignore ICMP echo datagrams and not respond to them; this is often a security measure to prevent certain kinds of Denial of Service attacks.

The ability to send ICMP datagrams may be restricted to users with administrative privileges, depending on the policies and configuration of the local system. If you are unable to send or receive any ICMP datagrams, it is recommended that you check the firewall settings and any third-party security software that could impact the normal operation of this component.

Example

```
BOOL bResult = pClient->TraceRoute(strAddress);  
if (bResult)  
{  
    ICMPTRACE icmpTrace;  
  
    bResult = pClient->GetFirstHost(icmpTrace);  
    while (bResult)  
    {  
        // The icmpTrace structure contains information about the
```

```
// intermediate host in the traceroute
CString strHostName;
pClient->ResolveAddress(icmpTrace.dwHostAddress, strHostName);

    bResult = pClient->GetNextHost(icmpTrace);
}
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[Echo](#), [GetFirstHost](#), [GetNextHost](#), [ResolveAddress](#)

Internet Control Message Protocol Data Structures

- ICMPTIME
- ICMPTRACE
- INTERNET_ADDRESS
- SYSTEMTIME

ICMPTIME Structure

This structure is used by the [GetTripTime](#) method to return the round-trip times of an ICMP datagram.

```
typedef struct _ICMPTIME {
    DWORD dwTripAverage;
    DWORD dwTripMaximum;
    DWORD dwTripMinimum;
    DWORD dwTripTime;
} ICMPTIME, *LPICMPTIME;
```

Members

dwTripAverage

The average round-trip time in milliseconds for all datagrams sent to the current host.

dwTripMaximum

The maximum round-trip time in milliseconds for all datagrams sent to current host.

dwTripMinimum

The minimum round-trip time in milliseconds for all datagrams sent to the current host.

dwTripTime

The current round-trip time in milliseconds for the last datagram sent to the current host.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

ICMPTRACE Structure

This structure is used by the [GetFirstHost](#) and [GetNextHost](#) methods to return the route of an ICMP datagram.

```
typedef struct _ICMPTRACE
{
    UINT        nDistance;
    DWORD       dwHostAddress;
    DWORD       dwTripAverage;
    DWORD       dwTripMaximum;
    DWORD       dwTripMinimum;
} ICMPTRACE, *LPICMPTRACE;
```

Members

nDistance

The distance from the local host to the remote host for this route.

dwHostAddress

An unsigned integer which specifies the IP address of the remote host in network byte order.

dwTripAverage

The average round-trip time in milliseconds for all datagrams sent to the specified host.

dwTripMaximum

The maximum round-trip time in milliseconds for all datagrams sent to specified host.

dwTripMinimum

The minimum round-trip time in milliseconds for all datagrams sent to the specified host.

dwTripTime

The current round-trip time in milliseconds for the last datagram sent to the specified host.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

INTERNET_ADDRESS Structure

This structure represents a numeric IPv4 or IPv6 address in network byte order.

```
typedef struct _INTERNET_ADDRESS
{
    INT    ipFamily;
    BYTE  ipNumber[16];
} INTERNET_ADDRESS, *LPINTERNET_ADDRESS;
```

Members

ipFamily

An integer which identifies the type of IP address. It will be one of the following values:

Constant	Description
INET_ADDRESS_UNKNOWN	The address has not been specified or the bytes in the <i>ipNumber</i> array does not represent a valid address. Functions which populate this structure will use this value to indicate that the address cannot be determined.
INET_ADDRESS_IPV4	Specifies that the address is in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address is in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

ipNumber

A byte array which contains the numeric form of the IP address. This array is large enough to store both IPv4 (32 bit) and IPv6 (128 bit) addresses. The values are stored in network byte order.

Remarks

The **INTERNET_ADDRESS** structure is used by some functions to represent an Internet address in a binary format that is compatible with both IPv4 and IPv6 addresses. Applications that use this structure should consider it to be opaque, and should not modify the contents of the structure directly.

For compatibility with legacy applications that expect an IP address to be 32 bits and stored in an unsigned integer, you can copy the first four bytes of the *ipNumber* array using the **CopyMemory** function or equivalent. Note that if this is done, your application should always check the *ipFamily* member first to make sure that it is actually an IPv4 address.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

SYSTEMTIME Structure

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include windows.h.

Internet Message Access Protocol Class Library

Manage email messages and mailboxes on a mail server.

Reference

- [Class Methods](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

Class Name	CImapClient
File Name	CSMAPV10.DLL
Version	10.0.1468.2518
LibID	E1419168-2803-4EF6-A0BC-48A71D9EEFD7
Import Library	CSMAPV10.LIB
Dependencies	None
Standards	RFC 3501

Overview

The Internet Message Access Protocol (IMAP) is an application protocol which is used to access a user's email messages which are stored on a mail server. However, unlike the Post Office Protocol (POP) where messages are downloaded and processed on the local system, the messages on an IMAP server are retained on the server and processed remotely. This is ideal for users who need access to a centralized store of messages or have limited bandwidth. For example, traveling salesmen who have notebook computers or mobile users on a wireless network would be ideal candidates for using IMAP.

The SocketTools IMAP library implements the current standard for this protocol, and provides methods to retrieve messages, or just certain parts of a message, create and manage mailboxes, search for specific messages based on certain criteria and so on. The API is designed as a superset of the Post Office Protocol API, so developers who are used to working with the POP3 library will find the IMAP library very easy to integrate into an existing application.

This library supports secure connections using the standard SSL and TLS protocols.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-

bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Internet Message Access Protocol Class Methods

Class	Description
CImapClient	Constructor which initializes the current instance of the class
~CImapClient	Destructor which releases resources allocated by the class
Method	Description
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified client handle to another thread
Cancel	Cancel the current blocking operation
CheckMailbox	Check the current mailbox for new messages
CloseMessage	Close the current message
Command	Send a command to the server
Connect	Connect to the specified server
CopyMessage	Copy a message from the current mailbox to another mailbox
CreateMailbox	Create a new mailbox on the server
CreateMessage	Create a new message in the specified mailbox
CreateSecurityCredentials	Allocate a structure to establish client security credentials
DeleteMailbox	Delete the specified mailbox from the server
DeleteMessage	Delete the specified message from the mailbox
DeleteSecurityCredentials	Delete the specified client security credentials
DetachHandle	Detach the handle for the current instance of this class
DisableEvents	Disable the event notification mechanism
DisableTrace	Disable logging of socket function calls to the trace log
Disconnect	Disconnect from the current server
EnableEvents	Enable the client event notification mechanism
EnableTrace	Enable logging of socket function calls to a file
EnumMessages	Enumerate the messages in the current mailbox
ExamineMailbox	Select the specified mailbox in read-only mode
ExpungeMailbox	Remove messages that have been marked for deletion from the current mailbox
FreezeEvents	Suspend and resume event handling by the client
GetCapability	Return a string which identifies the capabilities of the server
GetCurrentMailbox	Return the name of the currently selected mailbox
GetDeletedMessages	Return the messages that have been marked for deletion
GetErrorString	Return a description for the specified error code
GetFirstMailbox	Return the first mailbox according to specified criteria

GetHandle	Return the client handle used by this instance of the class
GetHeaderValue	Return the value of the specified header field
GetIdleThreadId	Return the ID of the thread created to monitor the client session
GetLastError	Return the last error code
GetMailboxDelimiter	Get the path delimiter for the specified mailbox hierarchy
GetMailboxSize	Return the size of the specified mailbox
GetMailboxStatus	Return the status of the specified mailbox
GetMailboxUID	Return the unique identifier for the specified mailbox
GetMessage	Retrieve the specified message from the server
GetMessageCount	Return the number of messages available in the mailbox
GetMessageFlags	Return the status flags for the specified message
GetMessageHeaders	Retrieve the specified message header from the server
GetMessageId	Return the message ID string for the specified message
GetMessageParts	Return the number of MIME message parts in the specified message
GetMessageSender	Return the address of the message sender
GetMessageSize	Return the size of the specified message
GetMessageUid	Return the unique identifier for the specified message
GetNewMessages	Return a list of the new messages in the current mailbox
GetNextMailbox	Return the next mailbox name on the server
GetResultCode	Return the result code from the previous command
GetResultString	Return the result string from the previous command
GetSecurityInformation	Return security information about the current client connection
GetStatus	Return the current status of the client
GetTimeout	Return the number of seconds until an operation times out
GetTransferStatus	Return data transfer statistics
GetUnseenMessages	Return a list of messages from the current mailbox that have not been read
Idle	Enables mailbox status monitoring for the client session
ImapEventProc	Callback method that processes events generated by the client
ImapIdleProc	Callback function that receives update notifications from the server
IsBlocking	Determine if the client is blocked, waiting for information
IsConnected	Determine if the client is connected to the server
IsInitialized	Determine if the class has been successfully initialized
IsReadable	Determine if data can be read from the server
IsWritable	Determine if data can be written to the server
Login	Login to the server
OpenMessage	Open the specified message for reading on the server

Read	Read data returned by the server
RegisterEvent	Register an event handler for the specified event
RenameMailbox	Rename the specified mailbox
ReselectMailbox	Reselect the current mailbox and return updated status information
SearchMailbox	Search the mailbox according to specified criteria
SelectMailbox	Select the specified mailbox in read-write mode
SetLastError	Set the last error code for the current thread
SetMessageFlags	Set one or more status flags for the specified message
SetTimeout	Set the number of seconds until an operation times out
ShowError	Display a message box with a description of the specified error
StoreMessage	Store the contents of a message to the specified file
SubscribeMailbox	Subscribe to the specified mailbox
UndeleteMessage	Undelete the specified message from the current mailbox
UnselectMailbox	Unselect the current mailbox and expunge any deleted messages
UnsubscribeMailbox	Unsubscribe from the specified mailbox
Write	Write data to the server

CImapClient::CImapClient Method

CImapClient();

The **CImapClient** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **ImapInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmav10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[~CImapClient](#), [IsInitialized](#)

CImapClient::~~CImapClient

`~CImapClient();`

The **CImapClient** destructor releases resources allocated by the current instance of the **CImapClient** object. It also uninitialized the library if there are no other concurrent uses of the class.

Remarks

When a **CImapClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmav10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CImapClient](#)

CIMapClient::AttachHandle Method

```
VOID AttachHandle(  
    HCLIENT hClient  
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle already has been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the *hClient* parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

See Also

[AttachThread](#), [DetachHandle](#), [GetHandle](#)

CImapClient::AttachThread Method

```
DWORD AttachThread(  
    DWORD dwThreadId  
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[AttachHandle](#), [Cancel](#), [Connect](#), [DetachHandle](#), [Disconnect](#), [GetHandle](#)

CImapClient::Cancel Method

```
INT Cancel();
```

The **Cancel** method cancels any outstanding blocking operation in the client, causing the blocking method to fail. The application may then retry the operation or terminate the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[IsBlocking](#)

CImapClient::CheckMailbox Method

```
INT CheckMailbox(  
    UINT *LpnMessages,  
    UINT *LpnUnseen  
);
```

The **CheckMailbox** method requests that the server create a checkpoint of the currently selected mailbox, and returns the current number of messages and unseen messages.

Parameters

lpnMessages

A pointer to an unsigned integer value which will contain the number of messages in the currently selected mailbox when the method returns.

lpnUnseen

A pointer to an unsigned integer value which will contain the number of unseen messages in the currently selected mailbox.

Return Value

If the method succeeds, it returns zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the client requests a checkpoint, the server may perform implementation-dependent housekeeping for that mailbox, such as updating the mailbox on disk with the current state of the mailbox in memory. On some systems this command has no effect other than to update the client with the current number of messages in the mailbox.

This method actually sends two IMAP commands. The first is the CHECK command, followed by the NOOP command to poll for any new messages that have arrived. In addition to polling the server for new messages, this command can also be used to ensure the idle timer on the server does not expire and force a disconnect from the client.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

See Also

[CreateMailbox](#), [GetFirstMailbox](#), [GetMailboxStatus](#), [GetNextMailbox](#), [RenameMailbox](#)

CImapClient::CloseMessage Method

```
INT CloseMessage();
```

The **CloseMessage** method closes the current message.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **CloseMessage** method closes the current message. If there is any remaining data left to be read from the message, it will be read and discarded.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmav10.lib

See Also

[CreateMessage](#), [OpenMessage](#)

CImapClient::Command Method

```
INT Command(  
    LPCTSTR LpszCommand,  
    LPCTSTR LpszParameter  
);
```

The **Command** method sends a command to the server. This method is typically used for site-specific commands not directly supported by the API.

Parameters

LpszCommand

The command which will be executed by the server.

LpszParameter

An optional command parameter. If the command requires more than one parameter, then they should be combined into a single string, with a space separating each parameter. If the command does not accept any parameters, this value may be NULL.

Return Value

If the method succeeds, it returns an IMAP result code. If the command was successful, it returns IMAP_RESULT_OK. A return value of IMAP_RESULT_CONTINUE indicates the command was accepted and the caller should proceed with the next command. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

A list of valid commands can be found in the technical specification for the protocol. The current version of the protocol which is supported by this library is version 4rev1 as defined in RFC 3501.

Use the **GetResultCode** method to determine the result of the command that was sent to the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetResultCode](#), [GetResultString](#)

CImapClient::Connect Method

```
BOOL Connect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **Connect** method is used to establish a connection with the server.

Parameters

lpszHostName

A pointer to a null terminated string which specifies the host name or IP address of the IMAP server.

nRemotePort

The port number the client should use to establish the connection. A value of zero specifies that default port 143 should be used, which is the standard port number assigned to the IMAP service. If the secure port number is specified, an implicit SSL/TLS connection will be established by default.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
IMAP_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
IMAP_OPTION_IDENTIFY	This option specifies the client should identify itself to the server. If enabled, the client will send the ID command to the server as defined in RFC 2971. This option has no effect if the server does not support the ID command.
IMAP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
IMAP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option

	only affects connections using either the SSL or TLS protocols.
IMAP_OPTION_SECURE	This option specifies that a secure connection should be established with the server and requires that the server support either the SSL or TLS protocol. This option is the same as specifying IMAP_OPTION_SECURE_EXPLICIT, which initiates the secure session using the STARTTLS command.
IMAP_OPTION_SECURE_EXPLICIT	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using either the SSL or TLS protocol and the STARTTLS command.
IMAP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using either the SSL or TLS protocol, and the secure session must be negotiated immediately after the connection has been established.
IMAP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
IMAP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
IMAP_OPTION_FREETHREAD	This option specifies that this instance of the class may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the class instance is synchronized across multiple threads.

hEventWnd

The handle to the asynchronous notification window. This window receives messages which notify the client of various asynchronous client events that occur. Specifying this parameter and a message identifier causes the connection to be non-blocking. If this parameter is NULL, then a blocking connection is established.

uEventMsg

The message identifier that is used when an asynchronous client event occurs. This value should

be greater than WM_USER as defined in the Windows header files. If the *hEventWnd* parameter is NULL, this parameter should be specified as WM_NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The use of Windows event notification messages has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

To prevent this method from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **Connect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

If you specify an event notification window, then the client session will be asynchronous. When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
IMAP_EVENT_CONNECT	The connection to the server has completed. The high word of the lParam parameter should be checked, since this notification message will be posted if an error has occurred.
IMAP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
IMAP_EVENT_READ	Data is available to read by the client. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the calling process is in asynchronous mode.
IMAP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
IMAP_EVENT_TIMEOUT	The client has timed out while waiting for a response from the server. Note that under some circumstances this event can be generated for a non-blocking connection, such as when the client is establishing a secure connection.
IMAP_EVENT_CANCEL	The client has canceled the current operation.
IMAP_EVENT_COMMAND	The client has processed a command that was sent to the server. The result code and result string can be used to determine if the response to the command. The high word of the lParam parameter should be checked, since this notification message will

	also be posed if the command cannot be executed.
IMAP_EVENT_PROGRESS	This event notification is sent periodically during lengthy blocking operations, such as retrieving a complete message from the server.

To cancel asynchronous notification and return the client to a blocking mode, use the **DisableEvents** method.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

The *dwOptions* argument can be used to specify the threading model that is used by the class when a connection is established. By default, the client session is initially attached to the thread that created it. From that point on, until the connection is terminated, only the owner may invoke methods in that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the IMAP_OPTION_FREETHREAD option enables any thread to call methods in any instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the class and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a method in the class, it must ensure that no other thread will call another method at the same time using the same instance.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableEvents](#), [Disconnect](#), [EnableEvents](#)

CImapClient::CopyMessage Method

```
INT CopyMessage(  
    UINT nMessageId,  
    LPCTSTR lpszMailbox  
);
```

The **CopyMessage** method copies a message from the current mailbox to the specified mailbox. The message is appended to the mailbox, and the message flags and internal date are preserved.

Parameters

nMessageId

The message identifier which specifies which message is to be copied to the mailbox. This value must be greater than zero and specify a valid message number.

lpszMailbox

A pointer to a string which specifies the name of the mailbox that the message will be copied to. The mailbox must already exist, and the client must have the appropriate access rights to modify the mailbox.

Return Value

If the method succeeds, it returns a value of zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the mailbox does not exist, the method will fail. To create a new mailbox, use the **CreateMailbox** method. A message can be copied within the same mailbox, in which case the server may flag it as a new message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateMailbox](#), [GetResultCode](#), [GetResultString](#)

CImapClient::CreateMailbox Method

```
INT CreateMailbox(  
    LPCTSTR lpszMailbox  
);
```

The **CreateMailbox** method creates a new mailbox on the server.

Parameters

lpszMailbox

A pointer to a string which specifies the new mailbox to be created.

Return Value

If the method succeeds, it returns a value of zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the mailbox name is suffixed with the server's hierarchy delimiter, this indicates to the server that the client intends to create mailbox names under the specified name in the hierarchy. If superior hierarchical names are specified in the mailbox name, then the server may automatically create them as needed. For example, if the mailbox name "Mail/Office/Projects" is specified and "Mail/Office" does not exist, it may be automatically created by the server.

The special mailbox name INBOX is reserved, and cannot be created. It is recommended that mailbox names only consist of printable ASCII characters, and the special characters "*" and "%" should be avoided.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DeleteMailbox](#), [GetFirstMailbox](#), [GetNextMailbox](#), [RenameMailbox](#)

CImapClient::CreateMessage Method

```
INT CreateMessage(  
    LPCTSTR lpszMailbox,  
    LPBYTE lpMessage,  
    DWORD dwMessageSize,  
    DWORD dwFlags  
);
```

```
INT CreateMessage(  
    LPCTSTR lpszMailbox,  
    LPCTSTR lpszMessage,  
    DWORD dwFlags  
);
```

The **CreateMessage** method creates a message, appending it to the contents of the specified mailbox.

Parameters

lpszMailbox

A pointer to a string which specifies the name of the mailbox that the message will be created in. The mailbox must already exist, and the client must have the appropriate access rights to modify the mailbox.

lpMessage

A pointer to the buffer which contains the data for the message to be created. An alternate form of the method accepts a pointer to a string.

dwMessageSize

An unsigned integer value which specifies the size of the message data in bytes.

dwFlags

An unsigned integer that specifies one or more message flags. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
IMAP_FLAG_NONE	No value.
IMAP_FLAG_ANSWERED	The message has been answered.
IMAP_FLAG_DRAFT	The message is not completed and is marked as a draft copy.
IMAP_FLAG_URGENT	The message is flagged for urgent or special attention.
IMAP_FLAG_SEEN	The message has been read.

Return Value

If the method succeeds, it returns zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the mailbox does not exist, the method will fail. To create a new mailbox, use the **CreateMailbox** method. If the message is created in the mailbox that is currently selected, the server may flag it as recent.

This method is typically used by applications to store messages which have already been sent to a user. After a message has been delivered using the SMTP protocol, that same message may be created in a mailbox on the IMAP server so that the user has access to those messages.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CopyMessage](#), [DeleteMessage](#), [GetMessageFlags](#), [SetMessageFlags](#)

CImapClient::CreateSecurityCredentials Method

```
BOOL CreateSecurityCredentials(  
    DWORD dwProtocol,  
    DWORD dwOptions,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName  
);  
  
BOOL CreateSecurityCredentials(  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName  
);  
  
BOOL CreateSecurityCredentials(  
    LPCTSTR lpszCertName  
);
```

The **CreateSecurityCredentials** method establishes the security credentials for the client session.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols.

	This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpzUserName

A pointer to a string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A pointer to a string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A pointer to a string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Example

```
pClient->CreateSecurityCredentials(  
    SECURITY_PROTOCOL_DEFAULT,  
    0,  
    NULL,  
    NULL,  
    lpszCertStore,  
    lpszCertName);
```

```
bConnected = pClient->Connect(lpszHostName,  
                               IMAP_PORT_SECURE,  
                               IMAP_TIMEOUT,  
                               IMAP_OPTION_SECURE);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [DeleteSecurityCredentials](#), [GetSecurityInformation](#), [SECURITYCREDENTIALS](#)

CImapClient::DeleteMailbox Method

```
INT DeleteMailbox(  
    LPCTSTR lpszMailbox  
);
```

The **DeleteMailbox** method deletes a mailbox on the server.

Parameters

lpszMailbox

A pointer to a string which specifies the mailbox to be deleted.

Return Value

If the method succeeds, it returns value of zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

A mailbox cannot be deleted if it contains inferior hierarchical names and has the IMAP_FLAG_NOSELECT attribute. On most systems this is the case when the mailbox name references a directory on the server, and that directory contains other subdirectories or mailboxes. To remove the mailbox, you must first delete any child mailboxes that exist.

If the mailbox that is deleted is the currently selected mailbox, it will be automatically unselected and any messages marked for deletion will be expunged before the mailbox is removed. If the delete operation fails, the client will remain in an unselected state until either the **ExamineMailbox** or **SelectMailbox** method is called.

The special mailbox name INBOX is reserved, and cannot be deleted.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateMailbox](#), [GetFirstMailbox](#), [GetNextMailbox](#), [RenameMailbox](#)

CImapClient::DeleteMessage Method

```
INT DeleteMessage(  
    UINT nMessageId  
);
```

The **DeleteMessage** method marks the specified message for deletion from the current mailbox.

Parameters

nMessage

Number of message to delete from the server. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method only marks the message for deletion. The message is not actually deleted until the mailbox is expunged or another mailbox is selected. This method will return an error if the current mailbox is in read-only mode, such as if it was selected using the **ExamineMailbox** method.

It is important to note that unlike the POP3 protocol, a message that is marked for deletion is still accessible on the IMAP server until the mailbox is expunged. This means, for example, that a deleted message can still be retrieved using the **GetMessage** method.

To determine if a message has been marked for deletion, use the **GetMessageFlags** method and check if the IMAP_FLAG_DELETED bit flag has been set. To list all of the deleted messages in the current mailbox, use the **GetDeletedMessages** method.

To remove the deletion flag from the message, use the **UndeleteMessage** method. To prevent all messages in the current mailbox from being expunged, use the **ReselectMailbox** method to reset the current mailbox state.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[GetDeletedMessages](#), [GetMessage](#), [GetMessageCount](#), [GetMessageFlags](#), [ReselectMailbox](#), [UndeleteMessage](#), [UnselectMailbox](#)

CImapClient::DeleteSecurityCredentials Method

```
VOID DeleteSecurityCredentials();
```

The **DeleteSecurityCredentials** method releases the security credentials for the current session.

Parameters

None.

Return Value

None.

Remarks

This method can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated. The security credentials are released when the class destructor is called, so it is normally not required that the application explicitly call this method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateSecurityCredentials](#)

CImapClient::DetachHandle Method

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmapi10.lib`

See Also

[AttachHandle](#), [GetHandle](#)

CImapClient::DisableEvents Method

```
INT DisableEvents();
```

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the client has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[EnableEvents](#), [RegisterEvent](#)

CImapClient::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmavp10.lib

See Also

[EnableTrace](#)

CImapClient::Disconnect Method

VOID Disconnect();

The **Disconnect** method terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmav10.lib

See Also

[Connect](#), [IsConnected](#)

CImapClient::EnableEvents Method

```
INT EnableEvents(  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Constant	Description
IMAP_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
IMAP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
IMAP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
IMAP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking

	operation. This event is only generated if the client is in asynchronous mode.
IMAP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
IMAP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
IMAP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
IMAP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

It is not required that the client be placed in asynchronous mode in order to receive command and progress event notifications. To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[DisableEvents](#), [RegisterEvent](#)

CImapClient::EnableTrace Method

```
BOOL EnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **EnableTrace** method enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace method logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableTrace](#)

CImapClient::EnumMessages Method

```
INT EnumMessages(  
    UINT nFirstMessageId,  
    UINT nLastMessageId,  
    DWORD dwMessageFlags,  
    LPIMAPMESSAGE LpMessageList,  
    INT nMaxMessages  
);  
  
INT EnumMessages(  
    LPIMAPMESSAGE LpMessageList,  
    INT nMaxMessages  
);
```

The **EnumMessages** method enumerates the messages in the current mailbox, populating an array of IMAPMESSAGE structures which contain information about each message.

Parameters

nFirstMessageId

An unsigned integer value which specifies the first message to enumerate. This value must be greater than zero and specify a valid message identifier. The first message in the mailbox has a value of one.

nLastMessageId

An unsigned integer value which specifies the last message to enumerate. This value must be greater than or equal to the value of the *nFirstMessageId* parameter. A special value of 0xFFFFFFFF (-1) can be used to specify the last message in the mailbox.

dwMessageFlags

This parameter is used to determine which messages are enumerated. If the value is zero, then all applicable messages will be enumerated. If the value is non-zero, only those messages which have at least one of the specified flags will be returned. More than one flag can be specified by using a bitwise operator. Valid message flags are:

Constant	Description
IMAP_FLAG_ANSWERED	Return only those messages which have been answered.
IMAP_FLAG_DELETED	Return only those messages which have been marked for deletion.
IMAP_FLAG_DRAFT	Return only those messages which have been marked as draft copies.
IMAP_FLAG_URGENT	Return only those messages which have been flagged for urgent or special attention.
IMAP_FLAG_RECENT	Return only those messages which have been recently added to the mailbox.
IMAP_FLAG_SEEN	Return only those messages which have been read.

LpMessageList

A pointer to an array of IMAPMESSAGE structures which will contain information about each of the messages returned by the server. This parameter cannot be NULL.

nMaxMessages

An integer value which specifies the maximum size of the IMAPMESSAGE array that was passed to the method. This value must be at least one.

Return Value

If the method succeeds, the return value is the number of messages that were enumerated. If no messages match the specified criteria, the method will return a value of zero. If an error is encountered, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the message UID is being stored locally by the client to identify the message over multiple sessions, it must also store the mailbox UID. Only the combination of the mailbox name, mailbox UID and message UID can be used to uniquely identify a given message on the server. Although IMAP server implementations are encouraged to maintain persistent message UIDs, they are not required to do so and those values may change if the mailbox UID changes.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

See Also

[GetMessage](#), [GetMessageCount](#), [GetMessageFlags](#), [GetMessageHeaders](#), [GetMessageSize](#), [GetMessageUid](#), [IMAPMESSAGE](#)

CImapClient::ExamineMailbox Method

```
INT ExamineMailbox(  
    LPCTSTR lpszMailbox,  
    LPIMAPMAILBOX lpMailboxInfo  
);
```

The **ExamineMailbox** method selects the specified mailbox for read-only access.

Parameters

lpszMailbox

A pointer to a string which specifies the new mailbox to be examined.

lpMailboxInfo

A pointer to an [IMAPMAILBOX](#) structure which contains information about the mailbox when the method returns. This parameter may be NULL if the caller does not require any information about the mailbox.

Return Value

If the method succeeds, it returns zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **ExamineMailbox** method is used to select a mailbox in read-only mode. Messages can be read, but they cannot be modified or deleted from the mailbox and new messages will not lose their status as new messages if they are accessed.

If the client has a different mailbox currently selected, that mailbox will be closed and any messages marked for deletion will be expunged. To prevent deleted messages from being removed from the previous mailbox, use the **UnselectMailbox** method prior to examining the new mailbox.

If an application wishes to update the information returned in the IMAPMAILBOX structure for the current mailbox, simply call **ExamineMailbox** again with the same mailbox name.

The special case-insensitive mailbox name INBOX is used for new messages. Other mailbox names may or may not be case-sensitive depending on the IMAP server's operating system and implementation.

To access a mailbox in read-write mode, use the **SelectMailbox** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DeleteMailbox](#), [GetFirstMailbox](#), [GetMailboxStatus](#), [GetNextMailbox](#), [RenameMailbox](#), [ReselectMailbox](#), [SelectMailbox](#), [UnselectMailbox](#)

CImapClient::ExpungeMailbox Method

```
INT ExpungeMailbox(  
    LPIMAPMAILBOX LpMailboxInfo  
);
```

The **ExpungeMailbox** method removes all messages marked for deletion from the current mailbox.

Parameters

LpMailboxInfo

A pointer to an [IMAPMAILBOX](#) structure which contains information about the mailbox when the method returns. This parameter may be NULL if the caller does not require any information about the mailbox.

Return Value

If the method succeeds, it returns zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **ExpungeMailbox** method causes all messages marked for deletion to be removed from the mailbox. Note that this can cause the mailbox's UID to change, and potentially invalidate the current message UIDs. It is recommended that applications use the information returned in the IMAPMAILBOX structure to update any internal state information stored on the local system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateMailbox](#), [ExamineMailbox](#), [GetFirstMailbox](#), [GetNextMailbox](#), [RenameMailbox](#), [SelectMailbox](#)

CImapClient::FreezeEvents Method

```
INT FreezeEvents(  
    BOOL bFreeze  
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the client.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableEvents](#), [EnableEvents](#), [RegisterEvent](#)

CImapClient::GetCapability Method

```
INT GetCapability(  
    LPTSTR lpszCapability,  
    INT nMaxLength  
);  
  
INT GetCapability(  
    CString& strCapability  
);
```

The **GetCapability** method returns a string which identifies the capabilities of the IMAP server.

Parameters

lpszCapability

A pointer to a null terminated string buffer that will contain one or more tokens separated by spaces which identify the capabilities of the IMAP server.

nMaxLength

The maximum length of the capability string, including the terminating null character.

Return Value

If the function succeeds, it returns the length of the capability string, not including the terminating null character. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method returns a string that contains one or more tokens separated by whitespace. Each token identifies a capability of the server. The following table lists some of the common capabilities:

Capability	Description
ACL	The RFC 2086 ACL extension
BINARY	The RFC 3516 binary content extension
CHILDREN	The RFC 3348 child mailbox extension
ID	The RFC 2971 ID extension
IDLE	The RFC 2177 IDLE extension
LOGINDISABLED	The RFC 2595 TLS/SSL extension
LOGINREFERRALS	The RFC 2221 login referrals extension
MAILBOXREFERRALS	The RFC 2193 mailbox referrals extension
MULTIAPPEND	The RFC 3501 MULTIAPPEND extension
NAMESPACE	The RFC 2342 namespace Extension
QUOTA	The RFC 2087 QUOTA extension
STARTTLS	The RFC 2595 TLS/SSL extension
UNSELECT	The RFC 3691 UNSELECT extension

Additional capabilities may be supported by your server. Note that experimental or custom

capabilities are always prefixed with the letter X. A list of standard IMAP capabilities is maintained by the Internet Assigned Numbers Authority (IANA) at www.iana.org.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmapi10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [Disconnect](#)

CImapClient::GetCurrentMailbox Method

```
INT GetCurrentMailbox(  
    LPTSTR lpszMailbox,  
    INT nMaxLength  
);  
  
INT GetCurrentMailbox(  
    CString& strMailbox  
);
```

The **GetCurrentMailbox** method returns the name of the current mailbox that has been selected.

Parameters

lpszMailbox

A pointer to a null terminated string buffer that will contain the current mailbox name. An alternate form of the method accepts a **CString** object which will contain the mailbox name when the method returns.

nMaxLength

The maximum length of the mailbox string, including the terminating null character.

Return Value

If the method succeeds, it returns the length of the current mailbox name, not including the terminating null character. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ExamineMailbox](#), [SelectMailbox](#), [UnselectMailbox](#)

CImapClient::GetDeletedMessages Method

```
INT GetDeletedMessages(  
    UINT* LpnMessageIds,  
    INT nMaxMessages  
);
```

The **GetDeletedMessages** method returns the message identifiers for those messages that have been marked for deletion in the current mailbox.

Parameters

lpnMessageIds

A pointer to an array of unsigned integers that will contain the message identifiers of those messages which have been marked for deletion in the current mailbox. This parameter may be NULL, in which case the method will return the number of deleted messages but will not return their identifiers.

nMaxMessages

The maximum number of message identifiers that may be stored in the *lpnMessageIds* array. If the *lpnMessageIds* parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of messages marked for deletion in the current mailbox. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The message identifiers returned by this method are only valid until the mailbox is expunged or another mailbox is selected.

To remove the deleted flag from a message, use the **UndeleteMessage** method. To prevent all messages in the current mailbox from being expunged, use the **ReselectMailbox** method to reset the current mailbox state.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

See Also

[GetMessageCount](#), [GetMessageFlags](#), [GetNewMessages](#), [GetUnseenMessages](#), [SearchMailbox](#)

CImapClient::GetErrorString Method

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);  
  
INT GetErrorString(  
    DWORD dwErrorCode,  
    CString& strDescription  
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLastError](#), [SetLastError](#), [ShowError](#)

CImapClient::GetFirstMailbox Method

```
INT GetFirstMailbox(  
    LPCTSTR lpszReference,  
    LPCTSTR lpszWildcard,  
    DWORD dwOptions,  
    LPTSTR lpszMailbox,  
    INT nMaxLength  
    LPDWORD lpdwFlags  
);  
  
INT GetFirstMailbox(  
    CString& strMailbox,  
    LPDWORD lpdwFlags  
);
```

The **GetFirstMailbox** method returns the name of the first matching mailbox.

Parameters

lpszReference

A pointer to a string which specifies the reference name. An empty string or NULL pointer specifies that the default mailbox hierarchy for the current user is returned. If the reference name is provided, this must be the name of a mailbox or a level of the mailbox hierarchy which provides the context in which the mailbox name is interpreted.

lpszWildcard

A pointer to a null terminated string which specifies the mailbox name to match. The wildcard character "*" may be used to match any portion of the mailbox hierarchy, including the delimiter. The wildcard character "%" matches any portion of the mailbox name, but does not match the mailbox delimiter. An empty string or NULL pointer specifies that all mailboxes in the context of the *lpszReference* parameter should be returned.

dwOptions

Specifies one or more options which controls how mailboxes are returned by the method. The options are bit flags which may be combined using a bitwise operator. One or more of the following values may be used:

Constant	Description
IMAP_LIST_DEFAULT	This option specifies that all regular, selectable mailboxes should be returned.
IMAP_LIST_SUBSCRIBED	This option specifies that only subscribed mailboxes should be returned.
IMAP_LIST_FOLDERS	This option specifies that non-selectable mailbox folders should also be returned.
IMAP_LIST_HIDDEN	This option specifies that hidden mailboxes should be returned.
IMAP_LIST_INFERIOR	This option specifies that inferior mailboxes should be returned if an explicit wildcard mask is not specified.

lpszMailbox

A pointer to a string buffer which will contain the first matching mailbox. This parameter cannot

be NULL. A minimum buffer size of at least 128 character is recommended.

nMaxLength

Specifies the maximum length of the string buffer. The maximum length of the buffer should be large enough to accommodate most path names on the IMAP server.

lpdwFlags

A pointer to an unsigned integer which will contain the mailbox flags for the first matching mailbox. This parameter may be NULL, in which case the mailbox flags are not returned. Otherwise, one or more of the following bit flags may be returned:

Constant	Description
IMAP_FLAG_NOINFERIORS	The mailbox does not contain any sub-mailboxes. In the IMAP protocol, these are referred to as inferior hierarchical mailbox names.
IMAP_FLAG_NOSELECT	The mailbox cannot be selected or examined. This flag is typically used by servers to indicate that the mailbox name refers to a directory on the server, not a mailbox file.
IMAP_FLAG_MARKED	The mailbox is marked as being of interest to a client. If this flag is used, it typically means that the mailbox contains messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.
IMAP_FLAG_UNMARKED	The mailbox is marked as not being of interest to a client. If this flag is used, it typically means that the mailbox does not contain any messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.

Return Value

If the method succeeds, it returns the length of the mailbox name. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetFirstMailbox** method is used to begin enumerating the available mailboxes for the current user on the IMAP server. Subsequent mailbox names are returned by calling **GetNextMailbox** until the method returns IMAP_ERROR with an error code of ST_ERROR_NO_MORE_MAILBOXES.

The method of the *lpdzReference* and *lpdzWildcard* parameters are implementation dependent and generally are tied to the underlying operating system. On a UNIX based system, it can be helpful to think of the reference name as the directory where mailbox folders are stored, and the mailbox name as the name to search for in that directory and any subdirectories, if applicable. If the reference name is an empty string or NULL pointer, this typically refers to the current user's home directory.

Generally speaking, a reference name should only be specified if you or the user of the application knows the directory structure on the IMAP server. Incorrectly using a reference name can have serious negative side-effects. For example, specifying a reference name of "/" on a UNIX based

system could cause the IMAP server to return search every directory on the system for a matching mailbox name. Similarly, the IMAP server may be unable to distinguish between regular files in the user's home directory and mailboxes. For this reason, most IMAP clients require that the user specify the directory on the server where their mailboxes are stored. Typically this is subdirectory named "mail" or "Mail" under the user's home directory. For non-UNIX servers, the mailbox hierarchy may be represented differently, including a flat hierarchy.

Hidden mailboxes are those mailboxes which use the UNIX convention of the name beginning with a period. Therefore, a mailbox named ".secrets" would not normally be returned by the **GetFirstMailbox** and **GetNextMailbox** methods. The IMAP_LIST_HIDDEN option causes all mailboxes to be returned.

The IMAP_LIST_INFERIOR option will return inferior mailboxes (mailboxes located in folders or subdirectories) if a wildcard mask is not specified. If a wildcard mask is specified, this option has no effect and only those mailboxes which match the wildcard will be returned.

Subscribed mailboxes are those which were specified using the **SubscribeMailbox** method. Marked mailboxes are typically those which have some special importance to the user.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DeleteMailbox](#), [GetMailboxStatus](#), [GetNextMailbox](#), [RenameMailbox](#), [SelectMailbox](#)

CImapClient::GetHandle Method

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmapi10.lib`

See Also

[AttachHandle](#), [DetachHandle](#), [IsInitialized](#)

CImapClient::GetHeaderValue Method

```
INT GetHeaderValue(  
    UINT nMessageId,  
    UINT nMessagePart,  
    LPCTSTR lpszHeader,  
    LPTSTR lpszValue,  
    INT nMaxLength  
);
```

```
INT GetHeaderValue(  
    UINT nMessageId,  
    LPCTSTR lpszHeader,  
    LPTSTR lpszValue,  
    INT nMaxLength  
);
```

```
INT GetHeaderValue(  
    UINT nMessageId,  
    UINT nMessagePart,  
    LPCTSTR lpszHeader,  
    CString& strValue  
);
```

```
INT GetHeaderValue(  
    UINT nMessageId,  
    LPCTSTR lpszHeader,  
    CString& strValue  
);
```

The **GetHeaderValue** method returns the value of a header field in the specified message.

Parameters

nMessageId

Number of message to retrieve header value from. This value must be greater than zero. The first message in the mailbox is message number one.

nMessagePart

The message part that the header value will be retrieved from. Message parts start with a value of one. A value of zero specifies that the RFC822 header field for the message will be used.

lpszHeader

Pointer to a string which specifies the message header to retrieve. The colon should not be included in this string.

lpszValue

Pointer to a string buffer that will contain the value of the specified message header.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, it returns the length of the header field value. If the header field is not present in the message, the method will return a value of zero. If the method fails, the return value

is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetHeaderValue** method returns the value of header field from the specified message. This allows an application to be able to easily determine the value of a header such as the sender, or the subject of the message. Any header field, including non-standard extensions, may be returned by this method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetMessageHeaders](#), [GetMessageId](#), [GetMessageSender](#)

CImapClient::GetIdleThreadId Method

```
DWORD GetIdleThreadId();
```

The **GetIdleThreadId** method returns ID of the thread that is monitoring the client session.

Parameters

None

Return Value

If the method succeeds, it returns the ID of the thread that is checking for update notifications from the server. If there is no active thread monitoring the client session, the method will return zero.

Remarks

The worker thread that monitors the client connection in the background can terminate if an IMAP command is sent to the server, if the **Cancel** method is called or if the client disconnects from the server. The **GetIdleThreadId** method enables the application to determine if this background thread is still active or not.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmavp10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Idle](#), [ImapIdleProc](#)

CImapClient::GetLastError Method

```
DWORD GetLastError();  
  
DWORD GetLastError(  
    CString& strDescription  
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError(0)** when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or IMAP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

See Also

[GetErrorString](#), [SetLastError](#), [ShowError](#)

CImapClient::GetMailboxDelimiter Method

```
INT GetMailboxDelimiter(  
    LPCTSTR lpszMailbox,  
    LPTSTR lpszDelimiter,  
    INT nMaxLength  
);  
  
INT GetMailboxDelimiter(  
    LPCTSTR lpszMailbox,  
    CString& strDelimiter  
);
```

The **GetMailboxDelimiter** method returns the hierarchical path delimiter used for the specified mailbox.

Parameters

lpszMailbox

A pointer to a null terminated string which specifies the mailbox name. This parameter may be NULL or an empty string, in which case the default delimiter will be returned.

lpszDelimiter

A pointer to a null terminated string buffer that will contain the mailbox delimiter. An alternate form of this method accepts a **CString** object that will contain the delimiter when the method returns.

nMaxLength

The maximum length of the delimiter string, including the terminating null character.

Return Value

If the method succeeds, it returns the length of the delimiter for the specified mailbox, not including the terminating null character. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the IMAP server supports multiple levels of mailboxes, then a special character or sequence of characters are used as delimiters between different levels of the mailbox hierarchy. On most systems, including most UNIX and Windows platforms, the delimiter is the forward slash "/" character.

It is possible that an IMAP server may only support a flat namespace, in which case this method will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ExamineMailbox](#), [SelectMailbox](#), [UnselectMailbox](#)

CImapClient::GetMailboxSize Method

```
DWORD GetMailboxSize(  
    LPCTSTR lpszMailbox  
);
```

The **GetMailboxSize** method returns the size of the specified mailbox.

Parameters

lpszMailbox

A pointer to a string which specifies the mailbox name.

Return Value

If the method succeeds, it returns the size of the mailbox. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMailboxSize** method may require a significant amount of time to calculate the mailbox size if there are a large number of messages in the mailbox. If the specified mailbox is not currently selected, then the current mailbox is unselected, the new mailbox is selected and the size calculated, and then the original mailbox is re-selected. This will have the side-effect of causing any messages marked for deletion to be expunged from the mailbox.

Because it can potentially result in long delays, it is not recommended that an application calculate the mailbox size unless it is absolutely necessary.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateMailbox](#), [GetFirstMailbox](#), [GetNextMailbox](#), [RenameMailbox](#)

CImapClient::GetMailboxStatus Method

```
INT GetMailboxStatus(  
    LPCTSTR lpszMailbox,  
    LPIMAPMAILBOXSTATUS lpMailboxStatus  
);
```

The **GetMailboxStatus** method returns status information about the specified mailbox.

Parameters

lpszMailbox

A pointer to a null terminated string that specifies the mailbox to return information about.

lpMailboxStatus

A pointer to an [IMAPMAILBOXSTATUS](#) structure which contains status information about the specified mailbox.

Return Value

If the method succeeds, it returns zero. If an error occurs, the method method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMailboxStatus** method enables an application to obtain status information about a mailbox without having to select another mailbox or open a second connection to the IMAP server to examine the mailbox. The information returned is a subset of the information returned when a mailbox is selected.

Note that obtaining status information for a mailbox may be a slow operation. It may require that server open the mailbox in read-only mode internally in order to obtain some of the status information. For this reason, this method should not be used to check for new messages; use the **CheckMailbox** method instead.

Some IMAP servers may return an error if you attempt to obtain status information about the currently selected mailbox. The protocol standard states that clients should not use this method on the currently selected mailbox, and should instead use the information returned by the **SelectMailbox** or **ExamineMailbox** methods.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CheckMailbox](#), [ExamineMailbox](#), [SelectMailbox](#)

CImapClient::GetMailboxUID Method

```
DWORD GetMailboxUID(  
    LPCTSTR lpszMailbox  
);
```

The **GetMailboxUID** method returns the unique identifier for the specified mailbox.

Parameters

lpszMailbox

A pointer to a null terminated string that specifies the mailbox name.

Return Value

If the method succeeds, it returns a non-zero value. If no unique identifier is assigned to the mailbox, the method will return zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMailboxUID** method returns an unsigned 32-bit value which uniquely identifies the mailbox and corresponds to the UIDVALIDITY value returned by the IMAP server. The actual value is determined by the server and should be considered opaque. The protocol specification requires that a mailbox's UID must not change unless the mailbox contents are modified or existing messages in the mailbox have been assigned new UIDs.

An application can use the mailbox UID value in combination with the message UID in order to uniquely identify a message on the server. However, the application must take into consideration that the IMAP server can reassign new message UIDs when the mailbox is modified. If the mailbox and message UIDs are being stored on the local system to track what messages have been retrieved from the server, the application must check the UID of the mailbox whenever it is selected. If the mailbox UID has changed, this means that the UIDs for the messages in that mailbox may have changed. The client should resynchronize with the server, and update its local copy of that mailbox.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetCurrentMailbox](#), [GetMailboxStatus](#), [GetMessageUid](#)

CImapClient::GetMessage Method

```
INT GetMessage(  
    UINT nMessageId,  
    UINT nMessagePart,  
    LPBYTE lpMessage,  
    LPDWORD lpdwLength,  
    DWORD dwOptions  
);
```

```
INT GetMessage(  
    UINT nMessageId,  
    UINT nMessagePart,  
    HGLOBAL* lpMessage,  
    LPDWORD lpdwLength,  
    DWORD dwOptions  
);
```

```
INT GetMessage(  
    UINT nMessageId,  
    UINT nMessagePart,  
    CString& strMessage,  
    DWORD dwOptions  
);
```

```
INT GetMessage(  
    UINT nMessageId,  
    CString& strMessage  
);
```

The **GetMessage** method retrieves a message from the server.

Parameters

nMessageId

Number of message to retrieve from the server. This value must be greater than zero. The first message in the mailbox is message number one.

nMessagePart

The message part that will be retrieved. A value of zero specifies that the complete message should be returned. If the message is a multipart MIME message, message parts start with a value of one.

lpMessage

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the method returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpMessage* parameter. If the *lpMessage* parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual length of the message that was downloaded.

dwOptions

An unsigned integer value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
IMAP_SECTION_DEFAULT	All headers and the complete body of the specified message or message part are to be retrieved. The client application is responsible for parsing the header block. If the message is a MIME multipart message and the complete message is returned, the application is responsible for parsing the individual message parts if necessary.
IMAP_SECTION_HEADER	All headers for the specified message or message part are to be retrieved. The client application is responsible for parsing the header block.
IMAP_SECTION_MIMEHEADER	The MIME headers for the specified message or message are to be retrieved. Only those header fields which are used in MIME messages, such as Content-Type will be returned to the client. This is typically useful when processing the header for a multipart message which contains file attachments. The client application is responsible for parsing the header block.
IMAP_SECTION_BODY	The body of the specified message or message part is to be retrieved. For a MIME formatted message, this may include data that is uuencoded or base64 encoded. The application is responsible for decoding this data.
IMAP_SECTION_PREVIEW	The message header or body is being previewed and should not be marked as read by the server. This prevents the message from having the IMAP_FLAG_SEEN flag from being automatically set when the message data is retrieved.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessage** method is used to retrieve a message from the server and copy it into a local buffer. The method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the message. In this case, the *lpMessage* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpMessage* parameter point to a global memory handle which will contain the message data when the method returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur.

This method will cause the current thread to block until the transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the IMAP_EVENT_PROGRESS event will be

periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine if a message is a multipart MIME message, use the **GetMessageParts** method. The return value specifies the number of parts in the message, with a value greater than one indicating that it is a multipart message. Combining the IMAP_SECTION_HEADER and IMAP_SECTION_BODY options will only return the header and body for the specified message if the *nMessagePart* parameter is zero. Due to a limitation of the IMAP FETCH command, if a message part is specified then only the body of that message part will be returned.

Note that unlike the SocketTools MIME API which considers the first message part to be zero, the IMAP protocol defines the first message part to be one.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[GetMessageCount](#), [GetMessageHeaders](#), [GetMessageParts](#), [StoreMessage](#)

CImapClient::GetMessageCount Method

```
INT GetMessageCount(  
    UINT *LpnLastMessage,  
    DWORD *LpdwMailboxSize  
);
```

The **GetMessageCount** method returns the number of messages that are available in the currently selected mailbox, and optionally the size of the mailbox in bytes.

Parameters

LpnLastMessage

Address of a variable that receives the number of the last valid message in the mailbox. If a NULL value is specified, this argument is ignored.

LpdwMailboxSize

Address of a variable that receives the current size of the mailbox. If a NULL value is specified, this argument is ignored.

Return Value

If the method succeeds, it returns the number of messages that are currently available. If no messages are available, this method will return zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessageCount** method is provided for compatibility with the POP3 class. The *LpnLastMessage* parameter will always contain the same value returned by the method since there is no distinction between the message count and the last available message. This is because messages that are marked for deletion on an IMAP server can still be accessed until the mailbox is expunged or unselected. This differs from the POP3 protocol, where messages cannot be accessed once they have been marked for deletion.

If the *LpdwMailboxSize* parameter is specified, this method will call **GetMailboxSize** to determine the size of the currently selected mailbox. Unlike the POP3 protocol, calculating the mailbox size may require a significant amount of time if there are a large number of messages in the mailbox. It is not recommended that an application request the mailbox size unless it is absolutely necessary.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DeleteMessage](#), [GetHeaderValue](#), [GetMailboxSize](#), [GetMessage](#), [GetMessageHeaders](#), [StoreMessage](#)

CImapClient::GetMessageFlags Method

```
BOOL GetMessageFlags(  
    UINT nMessageId,  
    LPDWORD lpdwMessageFlags  
);
```

The **GetMessageFlags** method returns the message flags for the specified message.

Parameters

nMessageId

Number of message to obtain the message flags for. This value must be greater than zero. The first message in the mailbox is message number one.

lpdwMessageFlags

Pointer to an unsigned integer that will contain the message flags for the specified message. The value may be zero, or one or more of the following values:

Constant	Description
IMAP_FLAG_ANSWERED	The message has been answered.
IMAP_FLAG_DELETED	The message has been marked for deletion.
IMAP_FLAG_DRAFT	The message has not been completed and is marked as a draft copy.
IMAP_FLAG_URGENT	The message has been flagged for urgent or special attention.
IMAP_FLAG_RECENT	The message has been added to the mailbox recently.
IMAP_FLAG_SEEN	The message has been read.

Return Value

If the method succeeds, the return value is non-zero and the *lpdwMessageFlags* parameter contains the status flags for the specified message. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

See Also

[DeleteMessage](#), [GetMessageCount](#), [SetMessageFlags](#)

CImapClient::GetMessageHeaders Method

```
INT GetMessageHeaders(  
    UINT nMessageId,  
    LPBYTE lpHeaders,  
    LPDWORD lpdwLength  
);
```

```
INT GetMessageHeaders(  
    UINT nMessageId,  
    HGLOBAL* lpHeaders,  
    LPDWORD lpdwLength  
);
```

```
INT GetMessageHeaders(  
    UINT nMessageId,  
    CString& strHeaders  
);
```

The **GetMessageHeaders** method retrieves the headers for the specified message from the server.

Parameters

nMessageId

Number of message to retrieve from the server. This value must be greater than zero. The first message in the mailbox is message number one.

lpHeaders

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the method returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpHeaders* parameter. If the *lpHeaders* parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual length of the message that was downloaded.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessageHeaders** method is used to retrieve a message header block from the server and copy it into a local buffer. The method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the *lpHeaders* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpHeaders* parameter point to a global memory handle which will contain the file data when the method returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur.

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the `IMAP_EVENT_PROGRESS` event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmapi10.lib`

See Also

[GetMessage](#), [GetMessageCount](#), [GetMessageParts](#), [StoreMessage](#)

CImapClient::GetMessageId Method

```
INT GetMessageId(  
    UINT nMessageId,  
    LPTSTR lpszMessageId,  
    INT nMaxLength  
);  
  
INT GetMessageId(  
    UINT nMessageId,  
    CString& strMessageId  
);
```

The **GetMessageId** method returns the message identifier string for the specified message.

Parameters

nMessageId

Number of message to retrieve the unique identifier for. This value must be greater than zero. The first message in the mailbox is message number one.

lpszMessageId

Address of a string buffer to receive the message identifier. This should be at least 64 bytes in length.

nMaxLength

The maximum length of the string buffer.

Return Value

If the method succeeds, the return value is the length of the unique identifier string. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessageId** method returns the message identifier from the Message-ID header of the specified message. The returned value is a string which can be used to identify a specific message, regardless if the message is moved to a different mailbox.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHeaderValue](#), [GetMessage](#), [GetMessageHeaders](#), [GetMessageSender](#)

CImapClient::GetMessageParts Method

```
INT GetMessageParts(  
    UINT nMessageId  
);
```

The **GetMessageParts** method returns the number of parts in a MIME multipart message on the server.

Parameters

nMessageId

Number of message to retrieve the part count for. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the method succeeds, the return value is the number of parts in the specified message. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessageParts** method can be used to determine if a message on the server contains multiple parts. A multipart MIME message typically contains file attachments or multiple representations of the message, such as a version of the message in plain text and another using HTML markup.

If the method returns a value of one, then the message does not contain multiple parts and is a standard RFC822 formatted message. A value greater than one indicates that the message does have multiple parts. The **GetMessageEx** method may be used to retrieve the data for a specific part of the message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[GetHeaderValue](#), [GetMessage](#), [GetMessageHeaders](#), [GetMessageId](#), [GetMessageSender](#)

CImapClient::GetMessageSender Method

```
INT GetMessageSender(  
    UINT nMessageId,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

```
INT GetMessageSender(  
    UINT nMessageId,  
    CString& strAddress  
);
```

The **GetMessageSender** method returns the sender's address for the specified message.

Parameters

nMessageId

Number of message to retrieve header value from. This value must be greater than zero. The first message in the mailbox is message number one.

lpszAddress

Pointer to a string buffer that will contain the address of the message sender.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, it returns the length of the address. If the sender cannot be determined, the method will return a value of zero. If the method fails, the return value is `IMAP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **GetMessageSender** method returns the email address specified in the Return-Path header field. This allows an application to be able to easily determine the sender, without parsing the header or downloading the contents of the message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmapi10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHeaderValue](#), [GetMessageHeaders](#), [GetMessageId](#)

CImapClient::GetMessageSize Method

```
DWORD GetMessageSize(  
    UINT nMessageId  
);
```

The **GetMessageSize** method returns the size of the specified message.

Parameters

nMessageId

Number of message to retrieve size of. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the method succeeds, the return value is the size of the specified message in bytes. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[GetHeaderValue](#), [GetMessageHeaders](#), [GetMessageId](#), [GetMessageSender](#)

CImapClient::GetMessageUid Method

```
DWORD GetMessageUid(  
    UINT nMessageId  
);
```

The **GetMessageUid** method returns the unique identifier (UID) for the specified message in the current mailbox.

Parameters

nMessageId

Number of message to obtain the unique identifier for. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the method succeeds, it returns a non-zero value. If no unique identifier is assigned to the message, the method will return zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessageUid** method returns an unsigned integer value which specifies a unique identifier for this message. The actual value is determined by the server and should be considered opaque.

An application can use the message UID value in combination with the mailbox UID in order to uniquely identify a message on the server. However, the application must take into consideration that the IMAP server can reassign new message UIDs when the mailbox is modified. If the mailbox and message UIDs are being stored on the local system to track what messages have been retrieved from the server, the application must check the UID of the mailbox whenever it is selected. If the mailbox UID has changed, this means that the UIDs for the messages in that mailbox may have changed. The client should resynchronize with the server, and update it's local copy of that mailbox.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[GetCurrentMailbox](#), [GetMailboxStatus](#), [GetMailboxUID](#)

CImapClient::GetNewMessages Method

```
INT GetNewMessages(  
    UINT* LpnMessageIds,  
    INT nMaxMessages  
);
```

The **GetNewMessages** method returns the message identifiers for those messages that have recently been added to the mailbox and have not been read.

Parameters

lpnMessageIds

A pointer to an array of unsigned integers that will contain the message identifiers of those messages that have recently been added to the mailbox and have not been read. This parameter may be NULL, in which case the method will return the number of new messages but will not return their identifiers.

nMaxMessages

The maximum number of message identifiers that may be stored in the *lpnMessageIds* array. If the *lpnMessageIds* parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of new messages in the current mailbox. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The message identifiers returned by this method are only valid until the mailbox is expunged or another mailbox is selected. Once a message has been read using **GetMessage** or **StoreMessage**, it is no longer considered to be a new message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

See Also

[GetDeletedMessages](#), [GetMessageCount](#), [GetMessageFlags](#), [GetUnseenMessages](#), [SearchMailbox](#)

CImapClient::GetNextMailbox Method

```
INT GetNextMailbox(  
    LPTSTR lpszMailbox,  
    INT nMaxLength  
    LPDWORD lpdwFlags  
);
```

```
INT GetNextMailbox(  
    CString& strMailbox,  
    LPDWORD lpdwFlags  
);
```

The **GetNextMailbox** method returns the name of the next matching mailbox.

Parameters

lpszMailbox

A pointer to a string buffer which will contain the next matching mailbox. This parameter cannot be NULL. A minimum buffer size of at least 128 character is recommended.

nMaxLength

Specifies the maximum length of the string buffer. The maximum length of the buffer should be large enough to accommodate most path names on the IMAP server.

lpdwFlags

A pointer to an unsigned integer which will contain the mailbox flags for the next matching mailbox. This parameter may be NULL, in which case the mailbox flags are not returned. Otherwise, one or more of the following bit flags may be returned:

Constant	Description
IMAP_FLAG_NOINFERIORS	The mailbox does not contain any sub-mailboxes. In the IMAP protocol, these are referred to as inferior hierarchical mailbox names.
IMAP_FLAG_NOSELECT	The mailbox cannot be selected or examined. This flag is typically used by servers to indicate that the mailbox name refers to a directory on the server, not a mailbox file.
IMAP_FLAG_MARKED	The mailbox is marked as being of interest to a client. If this flag is used, it typically means that the mailbox contains messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.
IMAP_FLAG_UNMARKED	The mailbox is marked as not being of interest to a client. If this flag is used, it typically means that the mailbox does not contain any messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.

Return Value

If the method succeeds, it returns the length of the mailbox name. If an error occurs, the method

returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetNextMailbox** method returns the next matching mailbox name. When the last mailbox has been returned, the next call to this method will result in an error, with the last error code set to ST_ERROR_NO_MORE_MAILBOXES.

Subscribed mailboxes are those which were specified using the **SubscribeMailbox** method. Marked mailboxes are typically those which have some special importance to the user.

For more information about enumerating the available mailboxes on the IMAP server, refer to the **GetFirstMailbox** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DeleteMailbox](#), [GetFirstMailbox](#), [GetMailboxStatus](#), [RenameMailbox](#), [SelectMailbox](#)

CImapClient::GetResultCode Method

```
INT GetResultCode();
```

The **GetResultCode** method reads the result code returned by the server in response to a command. The result code is a three-digit numeric code, and indicates if the operation succeeded, failed or requires additional action by the client.

Parameters

None.

Return Value

If the method succeeds, the return value is the result code. If the method fails, the return value is `IMAP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

Result codes are three-digit numeric values returned by the server. They may be broken down into the following ranges:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmav10.lib`

See Also

[Command](#), [GetResultString](#)

CImapClient::GetResultString Method

```
INT GetResultString(  
    LPTSTR lpszResult,  
    INT cbResult  
);
```

```
INT GetResultString(  
    CString& strResult  
);
```

The **GetResultString** method returns the last message sent by the server along with the result code.

Parameters

lpszResult

A pointer to the buffer that will contain the result string returned by the server. An alternate form of the method accepts a **CString** argument which will contain the result string returned by the server.

cbResult

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetResultString** method is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **GetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Command](#), [GetResultCode](#)

CImapClient::GetSecurityInformation Method

```
BOOL GetSecurityInformation(  
    LPSECURITYINFO LpSecurityInfo  
);
```

The **GetSecurityInformation** method returns security protocol, encryption and certificate information about the current client connection.

Parameters

LpSecurityInfo

A pointer to a [SECURITYINFO](#) structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value `SECURITY_PROTOCOL_NONE`.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;  
securityInfo.dwSize = sizeof(SECURITYINFO);  
  
if (pClient->GetSecurityInformation(&securityInfo))  
{  
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)  
    {  
        MessageBox(NULL, _T("The connection is not secure"),  
                    _T("Connection"), MB_OK);  
    }  
    else  
    {  
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)  
        {  
            MessageBox(NULL, _T("The connection is secure"),  
                        _T("Connection"), MB_OK);  
        }  
        else  
        {  
            MessageBox(NULL, _T("The server certificate not valid"),  
                        _T("Connection"), MB_OK);  
        }  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [CreateSecurityCredentials](#), [SECURITYINFO](#)

CImapClient::GetStatus Method

INT GetStatus();

The **GetStatus** method the current status of the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the client status code. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetStatus** method returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Constant	Description
0	IMAP_STATUS_UNUSED	No connection has been established.
1	IMAP_STATUS_IDLE	The client is current idle and not sending or receiving data.
2	IMAP_STATUS_CONNECT	The client is establishing a connection with the server.
3	IMAP_STATUS_READ	The client is reading data from the server.
4	IMAP_STATUS_WRITE	The client is writing data to the server.
5	IMAP_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[IsBlocking](#), [IsConnected](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#)

CImapClient::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

See Also

[Connect](#), [IsReadable](#), [IsWritable](#), [Read](#), [SetTimeout](#), [Write](#)

CImapClient::GetTransferStatus Method

```
INT GetTransferStatus(  
    LPIMAPTRANSFERSTATUS lpStatus  
);
```

The **GetTransferStatus** method returns information about the current file transfer in progress.

Parameters

lpStatus

A pointer to an [IMAPTRANSFERSTATUS](#) structure which contains information about the status of the current file transfer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetTransferStatus** method returns information about the current file transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no file currently being transferred, this method will return the status of the last successful transfer made by the client.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmavp10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableEvents](#), [GetStatus](#), [RegisterEvent](#)

CImapClient::GetUnseenMessages Method

```
INT GetUnseenMessages(  
    UINT * LpnMessageIds,  
    INT nMaxMessages  
);
```

The **GetUnseenMessages** method returns the message identifiers for those messages that have not been read.

Parameters

lpnMessageIds

A pointer to an array of unsigned integers that will contain the message identifiers of those messages that have not been read. This parameter may be NULL, in which case the method will return the number of unseen messages but will not return their identifiers.

nMaxMessages

The maximum number of message identifiers that may be stored in the *lpnMessageIds* array. If the *lpnMessageIds* parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of unseen messages in the current mailbox. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The message identifiers returned by this method are only valid until the mailbox is expunged or another mailbox is selected. Once a message has been read using **GetMessage** or **StoreMessage**, it is no longer considered to be an unseen message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[GetDeletedMessages](#), [GetMessageCount](#), [GetMessageFlags](#), [GetNewMessages](#), [SearchMailbox](#)

CImapClient::Idle Method

```
INT Idle(  
    IMAPIDLEPROC LpfnIdleProc,  
    DWORD_PTR dwParam  
);
```

```
INT Idle(  
    UINT nTimeout,  
    DWORD dwOptions,  
    IMAPIDLEPROC LpfnIdleProc,  
    DWORD_PTR dwParam  
);
```

The **Idle** method enables mailbox status monitoring for the client session, allowing the client to receive notifications from the server whenever a new message arrives or a message is expunged from the currently selected mailbox. This is typically used as an alternative to the client periodically polling the server for status information.

Parameters

nTimeout

Specifies the timeout period in seconds to wait for a notification from the server. This parameter is only used when the IMAP_IDLE_WAIT option has been specified.

dwOptions

Specifies the options which should be used when enabling idle monitoring. The following options are supported:

Constant	Description
IMAP_IDLE_NOWAIT	The method should return immediately after idle processing has been enabled. When this option is used, the application may continue to perform other methods while the client session is monitored for status updates sent by the server. The client will continue to monitor status changes until an IMAP command issued or the client disconnects from the server.
IMAP_IDLE_WAIT	The method should wait until the server sends a status update, or until the timeout period is reached. The client will stop monitoring status changes when the method returns. If this option is used in a single-threaded application, normal message processing can be impeded, causing the application to appear non-responsive until the timeout period is reached. It is strongly recommended that single-threaded applications with a user interface specify the IMAP_IDLE_NOWAIT option instead.

lpfnIdleProc

A pointer to an [IMAPIDLEPROC](#) callback function that will be invoked whenever the server sends an update notification to the client. This parameter must specify a valid function address and cannot be a NULL pointer.

dwParam

A user-defined value that is passed back to the caller whenever the callback function is invoked. This can be used to provide additional state information to the client. If it is not needed, the

caller should use a value of zero.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `IMAP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

Many IMAP servers support the ability to asynchronously send status updates to the client, rather than have the client periodically poll the server. The client enables this feature by calling **Idle** and providing the address of a callback function that will be invoked whenever the server sends an update notification to the client. Typically these updates inform the client that a new message has arrived or that a message has been expunged from the mailbox.

The **Idle** method can operate in two modes, based on the options specified by the caller. If the option `IMAP_IDLE_NOWAIT` is specified, the method begins monitoring the client session and returns control immediately to the caller. If the server sends a update notification, the callback function will be invoked with information about the status change. If the option `IMAP_IDLE_WAIT` is specified, the method will block waiting for the server to send a notification message to the client. The method will return when either a message is received or the timeout period is exceeded.

Sending an IMAP command to the server will cause the client to stop monitoring the session for status changes. To explicitly stop monitoring the session, use the **Cancel** method. To determine if the current client session is being monitored, use the **GetIdleThreadId** method. A non-zero return value indicates that the client session is idle and being monitored.

This method works by sending the `IDLE` command to the server and starting a worker thread which monitors the connection and looks for untagged responses issued by the server. Callbacks will be invoked for `EXISTS`, `EXPUNGE` and `RECENT` messages. Note that some servers may periodically send untagged `OK` messages to the client, indicating that the connection is still active; these messages are explicitly ignored. Because the monitoring is performed in a different thread, the callback function is invoked in the context of that thread. Client event notifications are disabled while inside the callback function, and the **Idle** method cannot be used to restart monitoring from within a callback function.

Applications should not perform any operation that takes a significant amount of time or updates the user interface from within the callback function. Instead, use flags or send application defined messages to indicate a change in state. For example, if the server sends a notification that a new email message has arrived, the application should not attempt to read the new message and update the user interface from within the callback function. Instead, it could use the **CWindow::PostMessage** method to send an application-defined message to the UI thread indicating the change in state. The application would have a message handler for that Windows message and update the user interface, indicating that a new message has arrived.

An application should never make an assumption about how a particular server may send update notifications to the client. Servers can be configured to use different intervals at which notifications are sent. For example, a server may send new message notifications immediately, but may periodically notify the client when a message has been expunged. Alternatively, a server may only send notifications at fixed intervals, in which case the client would not be notified of any new messages until the interval period is reached. It is not possible for a client to know what a particular server's update interval is. Applications that require that degree of control should not use **Idle** and should poll the server instead.

Example

```

// Begin monitoring the client session for status changes; when a new
// message arrives or a message is expunged, the UpdateHandler callback
// function will be invoked.
BOOL CUserApp::MonitorUpdates()
{
    INT nResult;

    nResult = m_imapClient.Idle(UpdateHandler, (DWORD)this);

    if (nResult == IMAP_ERROR)
    {
        m_imapClient.ShowError();
        return FALSE;
    }

    return TRUE;
}

// This function is called whenever the server notifies the client
// that a new message has arrived or a message has been expunged from
// the current mailbox
#define WM_APP_NEWMESSAGE (WM_APP+1)
#define WM_APP_EXPUNGED (WM_APP+2)

VOID CALLBACK CUserApp::UpdateHandler(HCLIENT hClient, UINT nUpdateId, UINT
nMessageId, DWORD_PTR dwParam)
{
    CUserApp *pThis = (CUserApp *)dwParam;

    switch (nUpdateId)
    {
    case IMAP_UPDATE_MESSAGE:
        {
            // Send a message indicating that a new message has arrived
            pThis->m_pMainWnd->PostMessage(WM_APP_NEWMESSAGE, nMessageId, 0);
        }
        break;

    case IMAP_UPDATE_EXPUNGE:
        {
            // Send a message indicating that a message has been expunged
            pThis->m_pMainWnd->PostMessage(WM_APP_EXPUNGED, nMessageId, 0);
        }
        break;
    }
}

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetIdleThreadId](#), [ImapIdleProc](#)

CImapClient::IsBlocking Method

BOOL IsBlocking();

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking method may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Cancel](#), [GetStatus](#), [IsConnected](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#), [Read](#), [Write](#)

CImapClient::IsConnected Method

```
BOOL IsConnected();
```

The **IsConnected** method is used to determine if the client is currently connected to a server.

Parameters

None.

Return Value

If the client is connected to a server, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmav10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#)

CImapClient::IsInitialized Method

BOOL IsInitialized();

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmavp10.lib

See Also

[CImapClient](#), [IsBlocking](#), [IsConnected](#)

CImapClient::IsReadable Method

```
BOOL IsReadable(  
    INT nTimeout,  
    LPDWORD lpdwAvail  
);
```

The **IsReadable** method is used to determine if data is available to be read from the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the method returns a non-zero value. If the client cannot read any data, the method returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmav10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsConnected](#), [IsInitialized](#), [IsWritable](#), [Write](#)

CImapClient::IsWritable Method

```
BOOL IsWritable(  
    INT nTimeout  
);
```

The **IsWritable** method is used to determine if data can be written to the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the method returns a non-zero value. If the client cannot write any data, the method returns zero.

Remarks

Although this method can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsConnected](#), [IsInitialized](#), [IsReadable](#), [Write](#)

CImapClient::Login Method

```
INT Login(  
    UINT nAuthType,  
    LPCTSTR LpszUserName,  
    LPCTSTR LpszPassword  
);
```

```
INT Login(  
    LPCTSTR LpszUserName,  
    LPCTSTR LpszPassword  
);
```

The **Login** method authenticates the specified user in on the server. This method must be called after the connection has been established, and before attempting to retrieve messages or perform any other method on the server.

Parameters

nAuthType

Identifies the type of authentication that should be used when the client logs in to the mail server. The following authentication methods are supported:

Constant	Description
IMAP_AUTH_LOGIN	Standard cleartext username and password is sent to the server. This authentication method is supported by all servers. Note that some servers may only support LOGIN authentication if a secure connection has been established.
IMAP_AUTH_PLAIN	Login using the PLAIN authentication mechanism as defined in RFC 4959. This authentication method is supported by most servers, although some may require that client establish a secure connection.
IMAP_AUTH_XOAUTH2	This authentication type will use the XOAUTH2 method to authenticate the client session. This authentication method does not require the user password, instead the <i>lpszPassword</i> parameter must specify the OAuth 2.0 bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this method will fail.
IMAP_AUTH_BEARER	This authentication type will use the OAUTHBEARER method to authenticate the client session as defined in RFC 7628. This authentication method does not require the user password, instead the <i>lpszPassword</i> parameter must specify the OAuth 2.0 bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this method will fail.
IMAP_AUTH_ANONYMOUS	Login using the anonymous Simple Authentication and Security Layer (SASL) mechanism as defined in RFC 4505.

If this authentication method is specified, the <i>lpszUserName</i> parameter should specify a name or email address acceptable to the mail server. The <i>lpszPassword</i> parameter is ignored and may be NULL.

lpszUserName

A null terminated string which specifies the user name to be used to authenticate the current client session.

lpszPassword

A null terminated string which specifies the password to be used when authenticating the current client session. If you are using the IMAP_AUTH_XOAUTH2 or IMAP_AUTH_BEARER authentication methods, this parameter is not a password, instead it specifies the bearer token provided by the mail service.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

In some cases, the user may be pre-authenticated by the server. In this case, the method will fail with the last error set to ST_ERROR_ALREADY_AUTHENTICATED. If a particular authentication method is not supported by the server, the last error will be set to ST_ERROR_INVALID_AUTHENTICATION_TYPE. For compatibility with the greatest number of servers, it is recommended that you use IMAP_AUTH_LOGIN as the authentication method.

You should only use an OAuth 2.0 authentication method if you understand the process of how to request the access token. Obtaining an access token requires registering your application with the mail service provider (e.g.: Microsoft or Google), getting a unique client ID associated with your application and then requesting the access token using the appropriate scope for the service. Obtaining the initial token will typically involve interactive confirmation on the part of the user, requiring they grant permission to your application to access their mail account.

The IMAP_AUTH_XOAUTH2 and IMAP_AUTH_BEARER authentication methods are similar, but they are not interchangeable. Both use an OAuth 2.0 bearer token to authenticate the client session, but they differ in how the token is presented to the server. It is currently preferable to use the XOAUTH2 method because it is more widely available and some service providers do not yet support the OAUTHBEARER method.

Your application should not store an OAuth 2.0 bearer token for later use. They have a relatively short lifespan, typically about an hour, and are designed to be used with that session. You should specify offline access as part of the OAuth 2.0 scope if necessary and store the refresh token provided by the service. The refresh token has a much longer validity period and can be used to obtain a new bearer token when needed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

CImapClient::OpenMessage Method

```
INT OpenMessage(  
    UINT nMessageId,  
    UINT nMessagePart,  
    DWORD dwOffset,  
    LPDWORD lpdwLength,  
    DWORD dwOptions  
);
```

```
INT OpenMessage(  
    UINT nMessageId,  
    DWORD dwOptions  
);
```

The **OpenMessage** method opens a message or a specific part of a multipart message in the current mailbox. The message data may also be limited a specific byte offset and length, which can be useful for previewing the contents.

Parameters

nMessageId

Number of message to retrieve. This value must be greater than zero. The first message in the mailbox is message number one.

nMessagePart

The message part that will be retrieved. A value of zero specifies that the complete message should be returned. If the message is a multipart MIME message, message parts start with a value of one.

dwOffset

The byte offset into the message. This parameter can be used in conjunction with the *lpdwLength* parameter to return a specific part of a message. A value of zero specifies the beginning of the message.

lpdwLength

A pointer to an unsigned integer value which should be initialized to the maximum number of bytes to be read, and will contain the size of the message when the method returns. To specify the entire message, from the offset specified by the *dwOffset* parameter to the end of the message, initialize the *lpdwLength* parameter to a value of -1. This parameter may be NULL if the message size is not needed.

dwOptions

The low order word of this parameter specifies how the message data will be returned. It may be one of the following values:

Constant	Description
IMAP_SECTION_DEFAULT	All headers and the complete body of the specified message or message part are to be retrieved. The client application is responsible for parsing the header block. If the message is a MIME multipart message and the complete message is returned, the application is responsible for parsing the

	individual message parts if necessary.
IMAP_SECTION_HEADER	All headers for the specified message or message part are to be retrieved. The client application is responsible for parsing the header block.
IMAP_SECTION_MIMEHEADER	The MIME headers for the specified message or message are to be retrieved. Only those header fields which are used in MIME messages, such as Content-Type will be returned to the client. This is typically useful when processing the header for a multipart message which contains file attachments. The client application is responsible for parsing the header block.
IMAP_SECTION_BODY	The body of the specified message or message part is to be retrieved. For a MIME formatted message, this may include data that is uuencoded or base64 encoded. The application is responsible for decoding this data.
IMAP_SECTION_PREVIEW	The message header or body is being previewed and should not be marked as read by the server. This prevents the message from having the IMAP_FLAG_SEEN flag from being automatically set when the message data is retrieved.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **OpenMessageEx** method uses the FETCH command to access the specified message on the server. The client can then use the **Read** method to read the contents of the message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CloseMessage](#), [GetMessage](#), [GetMessageHeaders](#), [OpenMessage](#), [Read](#), [StoreMessage](#)

CImapClient::Read Method

```
INT Read(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Read(  
    CString& strBuffer,  
    INT cbBuffer  
);
```

The **Read** method reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be copied. An alternate form of this method allows a **CString** variable to be passed and data read from the socket will be returned in that string.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When **Read** is called and the client is in non-blocking mode, it is possible that the method will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[EnableEvents](#), [IsBlocking](#), [IsReadable](#), [IsWritable](#), [RegisterEvent](#), [Write](#)

CImapClient::RegisterEvent Method

```
INT RegisterEvent(  
    UINT nEventId,  
    IMAPEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Constant	Description
IMAP_EVENT_CONNECT	The connection to the server has completed.
IMAP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
IMAP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
IMAP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
IMAP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
IMAP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
IMAP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
IMAP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **ImapEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **ImapEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This method is typically used to register an event handler that is invoked while a message is being retrieved. The IMAP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one IMAP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstdlib10.lib

See Also

[DisableEvents](#), [EnableEvents](#), [FreezeEvents](#), [ImapEventProc](#)

CImapClient::RenameMailbox Method

```
INT RenameMailbox(  
    LPCTSTR lpszOldMailbox,  
    LPCTSTR lpszNewMailbox  
);
```

The **RenameMailbox** method renames an existing mailbox.

Parameters

lpszOldMailbox

A pointer to a string which specifies the mailbox to be renamed.

lpszNewMailbox

A pointer to a string which specifies the new mailbox name.

Return Value

If the method succeeds, it returns zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **RenameMailbox** method renames an existing mailbox on the server. The new mailbox name cannot exist on the server, or the method will fail.

If the existing mailbox name contains inferior hierarchical names (mailboxes under the specified mailbox) then those mailboxes will also be renamed. For example, if the mailbox "Mail/Pictures" contains two mailboxes, "Personal" and "Work" and it is renamed to "Mail/Images" then the two mailboxes under it would be automatically renamed to "Mail/Images/Personal" and "Mail/Images/Work".

If the mailbox being renamed is the currently selected mailbox, the current mailbox will be unselected and any messages marked for deletion will be expunged. The new mailbox name will then automatically be re-selected. To prevent deleted messages from being removed from the mailbox prior to being renamed, use the **UnselectMailbox** method to unselect the current mailbox before calling **RenameMailbox**. Note that if the rename operation fails, the client may be left in an unselected state.

It is permitted to rename the special mailbox INBOX. In this case, the messages will be moved from the INBOX mailbox to the new mailbox. If the INBOX mailbox is currently selected, the new mailbox will not automatically be selected. INBOX will remain the selected mailbox.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateMailbox](#), [DeleteMailbox](#), [GetFirstMailbox](#), [GetNextMailbox](#)

CImapClient::ReselectMailbox Method

```
BOOL ReselectMailbox(  
    LPIMAPMAILBOX LpMailboxInfo  
);
```

The **ReselectMailbox** method reselects the current mailbox and returns updated information about the status of the mailbox.

Parameters

LpMailboxInfo

A pointer to an [IMAPMAILBOX](#) structure which contains updated information about the mailbox when the method returns. This parameter may be NULL if the caller does not require any information about the mailbox.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **ReselectMailbox** method forces the current mailbox to be reselected and returns updated information about the status of the mailbox. Deleted messages are not expunged from the mailbox and remain marked for deletion.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

See Also

[ExamineMailbox](#), [SelectMailbox](#), [UnselectMailbox](#)

CImapClient::SearchMailbox Method

```
LONG SearchMailbox(  
    LPCTSTR lpszCriteria,  
    LPCTSTR lpszCharacterSet,  
    UINT* lpnMessageIds,  
    LONG nMaxMessages  
);
```

```
LONG SearchMailbox(  
    LPCTSTR lpszCriteria,  
    UINT* lpnMessageIds,  
    LONG nMaxMessages  
);
```

The **SearchMailbox** method searches the current mailbox for messages that match the specified criteria, returning matching message identifiers.

Parameters

lpszCriteria

A pointer to a string which specifies the search criteria.

lpszCharacterSet

A pointer to a string which specifies the character set to use when searching the mailbox. If this parameter is NULL or an empty string, the default US-ASCII character set will be used.

dwReserved

A reserved parameter which should be set to the value 0.

lpnMessageIds

A pointer to an array of unsigned integers that will contain the message identifiers of those messages which match the search criteria in the current mailbox. This parameter may be NULL, in which case the method will return the number of matching messages but will not return their identifiers.

nMaxMessages

The maximum number of message identifiers that may be stored in the *lpnMessageIds* array. If the *lpnMessageIds* parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of messages that meet the search criteria in the current mailbox. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SearchMailbox** method is used to search a mailbox for messages which match a given criteria and return a list of the matching message identifiers. The search criteria is composed of one or more search keywords and an optional value to match against. String searches are not case sensitive and partial matches in the message are returned.

The following search keywords are recognized:

Keyword	Description
ANSWERED	Match those messages which have the IMAP_FLAG_ANSWERED flag

	set.
BCC <i>address</i>	Match those messages which contain the specified address in the BCC header field.
BEFORE <i>date</i>	Match those messages which were added to the mailbox prior to the specified date.
BODY <i>string</i>	Match those messages where the body contains the specified string.
CC <i>address</i>	Match those messages which contain the specified address in the CC header field.
DELETED	Match those messages which have the IMAP_FLAG_DELETED flag set.
DRAFT	Match those messages which have the IMAP_FLAG_DRAFT flag set.
FLAGGED	Match those messages which have the IMAP_FLAG_URGENT flag set.
FROM <i>address</i>	Match those messages which contain the specified address in the FROM header field.
HEADER <i>field string</i>	Match those messages which contain the string in the specified header field. If no string is specified, then all messages which contain the header will be matched.
LARGER <i>size</i>	Match those messages which are larger than the specified size in bytes.
NEW	Match those messages which have the IMAP_FLAG_RECENT flag set, but not the IMAP_FLAG_SEEN flag.
OLD	Match those messages which do not have the IMAP_FLAG_RECENT flag set.
ON <i>date</i>	Match those messages which were added on the specified date.
RECENT	Match those messages which have the IMAP_FLAG_RECENT flag set.
SEEN	Match those messages which have the IMAP_FLAG_SEEN flag set.
SENTBEFORE <i>date</i>	Match those messages whose Date header value is earlier than the specified date.
SENTON <i>date</i>	Match those messages whose Date header value is the same as the specified date.
SENTSINCE <i>date</i>	Match those messages whose Date header value is later than the specified date.
SINCE <i>date</i>	Match those messages added to the mailbox after the specified date.
SMALLER <i>size</i>	Match those messages which are smaller than the specified size in bytes.
SUBJECT <i>string</i>	Match those messages whose Subject header contains the specified string.
TEXT <i>string</i>	Match those messages whose headers or body contains the specified string.
TO <i>address</i>	Match those messages which contain the specified address in the TO header field.

UID <i>sequence</i>	Match those messages with unique identifiers in the sequence set.
UNANSWERED	Match those messages which do not have the IMAP_FLAG_ANSWERED flag set.
UNDELETED	Match those messages which do not have the IMAP_FLAG_DELETED flag set.
UNDRAFT	Match those messages which do not have the IMAP_FLAG_DRAFT flag set.
UNFLAGGED	Match those messages which do not have the IMAP_FLAG_URGENT flag set.
UNSEEN	Match those messages which do not have the IMAP_FLAG_SEEN flag set.

In addition to the listed keywords, the keyword NOT may prefix a keyword to return those messages which do not match the search criteria. For example, "NOT TO user@domain.com" would return those messages which were not addressed to user@domain.com.

If multiple search keywords are specified, the result is the intersection of all those messages which meet the search criteria. For example, a search criteria of "DELETED SINCE 1-Jan-2003" would return all those messages which are marked for deletion and were added to the mailbox after 1 January 2003.

Those search keywords which expect dates must be specified in format *dd-mmm-yyyy* where the month is the three letter abbreviation for the month name. Note that the internal date the message was added to the mailbox is not the same as the value of the Date header field in the message.

The UID keyword expects a one or more unique message identifiers. These values may provided as comma separated list, or a range delimited by a colon. For example, "UID 23000:24000" would return all those messages who have UIDs ranging from 23000 through to 24000.

The message identifiers returned by this method are only valid until the mailbox is expunged or another mailbox is selected.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetDeletedMessages](#), [GetMessageCount](#), [GetMessageFlags](#), [GetNewMessages](#), [GetUnseenMessages](#)

CImapClient::SelectMailbox Method

```
INT SelectMailbox(  
    LPCTSTR lpszMailbox,  
    LPIMAPMAILBOX lpMailboxInfo  
);
```

The **SelectMailbox** method selects the specified mailbox for read-write access.

Parameters

lpszMailbox

A pointer to a string which specifies the new mailbox to be selected.

lpMailboxInfo

A pointer to an [IMAPMAILBOX](#) structure which contains information about the mailbox when the method returns. This parameter may be NULL if the caller does not require any information about the mailbox.

Return Value

If the method succeeds, it returns zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SelectMailbox** method is used to select a mailbox in read-write mode. If the client has a different mailbox currently selected, that mailbox will be closed and any messages marked for deletion will be expunged. To prevent deleted messages from being removed from the previous mailbox, use the **UnselectMailbox** method prior to selecting the new mailbox.

If an application wishes to update the information returned in the IMAPMAILBOX structure for the current mailbox, simply call **SelectMailbox** again with the same mailbox name. Note that this will not cause any messages marked for deletion to be expunged.

The special case-insensitive mailbox name INBOX is used for new messages. Other mailbox names may or may not be case-sensitive depending on the IMAP server's operating system and implementation.

To access a mailbox in read-only mode, use the **ExamineMailbox** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DeleteMailbox](#), [ExamineMailbox](#), [GetFirstMailbox](#), [GetMailboxStatus](#), [GetNextMailbox](#), [RenameMailbox](#), [ReselectMailbox](#), [UnselectMailbox](#)

CImapClient::SetLastError Method

```
VOID SetLastError(  
    DWORD dwErrorCode  
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or IMAP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmav10.lib

See Also

[GetErrorString](#), [GetLastError](#), [ShowError](#)

CImapClient::SetMessageFlags Method

```
INT SetMessageFlags(  
    UINT nMessageId,  
    UINT nMode  
    DWORD dwMessageFlags  
);
```

The **SetMessageFlags** method returns the message flags for the specified message.

Parameters

nMessageId

Number of message to obtain the message flags for. This value must be greater than zero. The first message in the mailbox is message number one.

nMode

An unsigned integer value which specifies one of the following modes which determines how the message flags are set:

Constant	Description
IMAP_FLAGS_REPLACE	All message flags are replaced with the flags specified by the <i>dwMessageFlags</i> parameter.
IMAP_FLAGS_ADD	The message flags specified by the <i>dwMessageFlags</i> parameter will be set for the message. Message flags that have been previously set will remain unmodified.
IMAP_FLAGS_REMOVE	The message flags specified by the <i>dwMessageFlags</i> parameter will be removed from the message. Message flags that are not specified will remain unmodified.

dwMessageFlags

An unsigned integer value which specifies one or more message flags. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
IMAP_FLAG_ANSWERED	The message has been answered.
IMAP_FLAG_DELETED	The message has been marked for deletion.
IMAP_FLAG_DRAFT	The message has not been completed and is marked as a draft copy.
IMAP_FLAG_URGENT	The message has been flagged for urgent or special attention.
IMAP_FLAG_RECENT	The message has been added to the mailbox recently.
IMAP_FLAG_SEEN	The message has been read.

Return Value

If the method succeeds, it returns IMAP_RESULT_OK. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[DeleteMessage](#), [GetMessageCount](#), [GetMessageFlags](#)

CImapClient::SetTimeout Method

```
INT SetTimeout(  
    UINT nTimeout  
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[Connect](#), [GetTimeout](#), [IsReadable](#), [IsWritable](#), [Read](#), [Write](#)

CImapClient::ShowError Method

```
INT ShowError(  
    LPCTSTR lpszAppTitle,  
    UINT uType,  
    DWORD dwErrorCode  
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A pointer to a string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetErrorString](#), [GetLastError](#), [SetLastError](#)

CImapClient::StoreMessage Method

```
INT StoreMessage(  
    UINT nMessageId,  
    LPCTSTR lpszFileName  
);
```

The **StoreMessage** method retrieves a message from the current mailbox and stores it in a local file or the system clipboard.

Parameters

nMessageId

Number of the message to retrieve. This value must be greater than zero. The first message in the mailbox is message number one.

lpszFileName

Pointer to a string which specifies the file that the message will be stored in. If an empty string or NULL pointer is passed as an argument, the message is copied to the system clipboard.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **StoreMessage** method provides a method of retrieving and storing a message on the local system. The contents of the message is stored as a text file, using the specified file name. This method always causes the caller to block until the entire message has been retrieved, even if the client has been put in asynchronous mode.

If event handling is enabled, the IMAP_EVENT_PROGRESS event will fire periodically during the transfer of the message to the local system. An application can determine how much of the message has been retrieved by calling the **GetTransferStatus** method.

To retrieve the message into a global memory buffer so that it can be passed to the MIME or SMTP libraries, use the **GetMessage** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetMessage](#), [GetMessageHeaders](#), [GetTransferStatus](#)

CImapClient::SubscribeMailbox Method

```
INT SubscribeMailbox(  
    LPCTSTR lpszMailbox  
);
```

The **SubscribeMailbox** method subscribes the user to the specified mailbox.

Parameters

lpszMailbox

A pointer to a string which specifies the mailbox to subscribe to.

Return Value

If the method succeeds, it returns zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SubscribeMailbox** method adds the specified mailbox to the current user's list of active or subscribed mailboxes. The user will remain subscribed to the mailbox across multiple sessions, until the **UnsubscribeMailbox** method is called to remove the mailbox from the subscription list.

To list those mailboxes which the user has subscribed to, use the **GetFirstMailbox** method and specify the IMAP_LIST_SUBSCRIBED option.

Note that if a user subscribes to a mailbox and that mailbox is later renamed or deleted, the mailbox will not be automatically removed from the user's subscription list. An application must not assume that because a mailbox name is included in the list of subscribed mailboxes, it exists and can be selected. To check if the mailbox exists, use the **GetMailboxStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ExamineMailbox](#), [GetFirstMailbox](#), [GetNextMailbox](#), [SelectMailbox](#), [UnselectMailbox](#), [UnsubscribeMailbox](#)

CImapClient::UndeleteMessage Method

```
INT UndeleteMessage(  
    UINT nMessageId  
);
```

The **UndeleteMessage** method removes the deletion flag for the specified message.

Parameters

nMessage

Number of message to undelete from the server. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmav10.lib

See Also

[DeleteMessage](#), [GetDeletedMessages](#), [GetMessage](#), [GetMessageCount](#), [GetMessageFlags](#), [ReselectMailbox](#), [UnselectMailbox](#)

CImapClient::UnselectMailbox Method

```
INT UnselectMailbox(  
    BOOL bExpunge  
);
```

The **UnselectMailbox** method unselects the current mailbox.

Parameters

bExpunge

A boolean flag which determines if deleted messages will be expunged from the mailbox. A non-zero value specifies that messages that have been marked for deletion will be removed from the mailbox. A zero value specifies that no messages will be removed from the mailbox.

Return Value

If the method succeeds, it returns zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[DeleteMailbox](#), [ExamineMailbox](#), [GetFirstMailbox](#), [GetMailboxStatus](#), [GetNextMailbox](#), [RenameMailbox](#), [ReselectMailbox](#), [SelectMailbox](#)

CImapClient::UnsubscribeMailbox Method

```
INT UnsubscribeMailbox(  
    LPCTSTR lpszMailbox  
);
```

The **UnsubscribeMailbox** method unsubscribes the user from the specified mailbox.

Parameters

lpszMailbox

A pointer to a string which specifies the mailbox to unsubscribe from.

Return Value

If the method succeeds, it returns zero. If an error occurs, the method returns IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **UnsubscribeMailbox** method removes the specified mailbox from the current user's list of active or subscribed mailboxes.

To list those mailboxes which the user has subscribed to, use the **GetFirstMailbox** method and specify the IMAP_LIST_SUBSCRIBED option.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ExamineMailbox](#), [GetFirstMailbox](#), [GetNextMailbox](#), [SelectMailbox](#), [SubscribeMailbox](#), [UnselectMailbox](#)

CImapClient::Write Method

```
INT Write(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Write(  
    LPCTSTR lpszBuffer  
    INT cbBuffer  
);
```

The **Write** method sends the specified number of bytes to the server.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server. In an alternate form of the method, the pointer is to a string.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero, unless a pointer to a string is passed as the buffer argument. In that case, if the value is -1, all of the characters in the string, up to but not including the terminating null character, will be sent to the server.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is IMAP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the next asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableEvents](#), [IsBlocking](#), [IsReadable](#), [IsWritable](#), [Read](#), [RegisterEvent](#)

Internet Message Access Protocol Data Structures

- IMAPMAILBOX
- IMAPMAILBOXSTATUS
- IMAPMESSAGE
- IMAPTRANSFERSTATUS
- SECURITYCREDENTIALS
- SECURITYINFO
- SYSTEMTIME

IMAPMAILBOX Structure

This structure contains information about a selected mailbox.

```
typedef struct _IMAPMAILBOX
{
    UINT    nMessages;
    UINT    nRecentMessages;
    UINT    nUnseenMessageId;
    DWORD   dwMailboxUID;
    DWORD   dwNextMessageUID;
    DWORD   dwFlags;
    DWORD   dwPermanentFlags;
    DWORD   dwAccessMode;
    DWORD   dwReserved1;
    DWORD   dwReserved2;
} IMAPMAILBOX, *LPIMAPMAILBOX;
```

Members

nMessages

A value specifies the total number of messages in the mailbox.

nRecentMessages

A value which specifies the number of new messages that have recently arrived in the mailbox.

nUnseenMessageId

A value which specifies the message ID of the first unseen message in the mailbox.

dwMailboxUID

A value which specifies a unique identifier for this mailbox which corresponds to the UIDVALIDITY value returned by the IMAP server. The actual value is determined by the server and should be considered opaque. The protocol specification requires that a mailbox's UID must not change unless the mailbox contents are modified or existing messages in the mailbox have been assigned new UIDs.

dwNextMessageUID

A value which specifies the predicted unique identifier that will be assigned to a new message in the mailbox. This corresponds to the UIDNEXT value returned by the IMAP server. The protocol specification requires that as long as the mailbox UID is unchanged, messages that are added to the mailbox will be assigned a UID greater than or equal to the next UID value.

dwFlags

A value which specifies one or more mailbox flags. One or more of the following values may be specified:

Constant	Description
IMAP_FLAG_NOINFERIORS	The mailbox does not contain any child mailboxes. In the IMAP protocol, these are referred to as inferior hierarchical mailbox names.
IMAP_FLAG_NOSELECT	The mailbox cannot be selected or examined. This flag is typically used by servers to indicate that the mailbox name refers to a directory on the server, not a mailbox file.
IMAP_FLAG_MARKED	The mailbox is marked as being of interest to a client. If

	this flag is used, it typically means that the mailbox contains messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.
IMAP_FLAG_UNMARKED	The mailbox is marked as not being of interest to a client. If this flag is used, it typically means that the mailbox does not contain any messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.

dwPermanentFlags

A value which specifies the message flags that a client can change permanently. If this value is zero, then no permanent flags are defined for the mailbox and the client may assume that all message flags may be set permanently. Otherwise, one or more of the following values may be specified:

Constant	Description
IMAP_FLAG_ANSWERED	The message has been answered.
IMAP_FLAG_DRAFT	The message has not been completed and is marked as a draft copy.
IMAP_FLAG_URGENT	The message has been flagged for urgent or special attention.
IMAP_FLAG_SEEN	The message has been read.

dwAccessMode

A value which specifies the access mode for the mailbox. It may be one of the following values:

Constant	Description
IMAP_ACCESS_READONLY	The mailbox has been selected in read-only mode. Messages may not be created in the mailbox, nor can message flags be modified.
IMAP_ACCESS_READWRITE	The mailbox has been selected in read-write mode. Messages may be modified by the client, and messages marked for deletion can be expunged.

dwReserved1

A reserved value that is undefined.

dwReserved2

A reserved value that is undefined.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

IMAPMAILBOXSTATUS Structure

This structure contains information about a mailbox.

```
typedef struct _IMAPMAILBOXSTATUS
{
    UINT    nMessages;
    UINT    nRecentMessages;
    UINT    nUnseenMessages;
    DWORD   dwMailboxUID;
    DWORD   dwNextMessageUID;
    DWORD   dwReserved;
} IMAPMAILBOXSTATUS, *LPIMAPMAILBOXSTATUS;
```

Members

nMessages

A value specifies the total number of messages in the mailbox.

nRecentMessages

A value which specifies the number of new messages that have recently arrived in the mailbox.

nUnseenMessages

A value which specifies the number of unread messages in the mailbox.

dwMailboxUID

A value which specifies a unique identifier for this mailbox which corresponds to the UIDVALIDITY value returned by the IMAP server. The actual value is determined by the server and should be considered opaque. The protocol specification requires that a mailbox's UID must not change unless the mailbox contents are modified or existing messages in the mailbox have been assigned new UIDs.

dwNextMessageUID

A value which specifies the predicted unique identifier that will be assigned to a new message in the mailbox. This corresponds to the UIDNEXT value returned by the IMAP server. The protocol specification requires that as long as the mailbox UID is unchanged, messages that are added to the mailbox will be assigned a UID greater than or equal to the next UID value.

dwReserved

A reserved value that is undefined.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

IMAPMESSAGE Structure

This structure contains information about a message.

```
typedef struct _IMAPMESSAGE
{
    UINT    nMessageId;
    DWORD   dwMessageUID;
    DWORD   dwSize;
    DWORD   dwFlags;
    DWORD   dwTimestamp;
    DWORD   dwReserved;
} IMAPMESSAGE, *LPIMAPMESSAGE;
```

Members

nMessageId

An integer value which identifies the message. The message identifier is only valid while the mailbox is selected and no messages marked for deletion have been expunged. To maintain a persistent identifier for the message, use a combination of the mailbox UID and message UID.

dwMessageUID

An integer value which specifies a unique identifier for this message. The actual value is determined by the server and should be considered opaque. If the client application stores the message UID on the local system, it should also store the UID for the mailbox that contains the message. If the mailbox UID changes, the message UID may no longer be valid.

dwSize

Specifies the size of the message in bytes.

dwFlags

A value which specifies one or more message flags. One or more of the following values may be specified:

Constant	Description
IMAP_FLAG_NONE	No value.
IMAP_FLAG_ANSWERED	The message has been answered.
IMAP_FLAG_DELETED	The message has been marked for deletion.
IMAP_FLAG_DRAFT	The message has not been completed and is marked as a draft copy.
IMAP_FLAG_URGENT	The message has been flagged for urgent or special attention.
IMAP_FLAG_RECENT	The message has been added to the mailbox recently.
IMAP_FLAG_SEEN	The message has been read.

dwTimestamp

An integer value which specifies the date and time that the message was created in the mailbox. The value is expressed as the number of seconds since midnight, 1 January 1970 and is the same value that is used for the standard C runtime library time methods. Note that the date and time used is the message's internal date from the mail server, not the value of the Date header field.

dwReserved

A reserved value that is undefined.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

IMAPTRANSFERSTATUS Structure

This structure is used by the [GetTransferStatus](#) method to return information about a file transfer in progress.

```
typedef struct _IMAPTRANSFERSTATUS
{
    UINT    nMessageId;
    DWORD   dwBytesTotal;
    DWORD   dwBytesCopied;
    DWORD   dwBytesPerSecond;
    DWORD   dwTimeElapsed;
    DWORD   dwTimeEstimated;
} IMAPTRANSFERSTATUS, *LPIMAPTRANSFERSTATUS;
```

Members

nMessageId

The message ID of the current message that is being transferred.

dwBytesTotal

The total number of bytes that will be transferred. If the file is being copied from the server to the local host, this is the size of the remote file. If the file is being copied from the local host to the server, it is the size of the local file. If the file size cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the file transfer is completed. This is based on the average number of bytes transferred per second.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

SECURITYCREDENTIALS Structure

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on

	what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that

will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A pointer to a null terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A pointer to a null terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A pointer to a null terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A pointer to a null terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
------------	-------------

CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
----	---

MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
----	--

ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.
------	--

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

A pointer to a null terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used for SSH connections

and should always be NULL when establishing a secure connection using SSL or TLS.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* member should be set to CREDENTIAL_STORE_FILENAME and the *lpzCertStore* member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS#12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the *lpzPassword* member or the library will be unable to access the certificate.

Note that the *lpzUserName* and *lpzPassword* members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

The TLS 1.1 and TLS 1.2 protocols are only supported on Windows 7, Windows Server 2008 R2 and later versions of the platform. If these options are specified and the application is running on Windows XP or Windows Vista, the protocol version will be downgraded to TLS 1.0 for backwards compatibility with those platforms.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include ctools10.h.

Unicode: Implemented as Unicode and ANSI versions.

SECURITYINFO Structure

This structure contains information about a secure connection that has been established.

```
typedef struct _SECURITYINFO
{
    DWORD        dwSize;
    DWORD        dwProtocol;
    DWORD        dwCipher;
    DWORD        dwCipherStrength;
    DWORD        dwHash;
    DWORD        dwHashStrength;
    DWORD        dwKeyExchange;
    DWORD        dwCertStatus;
    SYSTEMTIME   stCertIssued;
    SYSTEMTIME   stCertExpires;
    LPCTSTR      lpszCertIssuer;
    LPCTSTR      lpszCertSubject;
    LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to **sizeof(SECURITYINFO)** prior to passing the address of this structure to the function. Note that if this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the protocol that was selected to establish the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The

	<p>correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.</p>
SECURITY_PROTOCOL_TLS10	<p>The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS11	<p>The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS12	<p>The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.</p>

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_CIPHER_RC2	The RC2 block cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
SECURITY_CIPHER_DES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher, using 56-bit

	keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher (also known as the Rijndael cipher) is a fixed block size cipher which use a key size of 128, 192 or 256 bits. This cipher is supported on Windows XP SP3 and later versions of the operating system.
SECURITY_CIPHER_SKIPJACK	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
SECURITY_CIPHER_BLOWFISH	The Blowfish block cipher was selected. This is a variable key length cipher up to 448 bits, using a 64-bit block size.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected. One of the following values will be returned:

Constant	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected. One of the following values will be returned:

--	--

Constant	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography. This key exchange algorithm is only supported on Windows XP SP3 and later versions of the operating system.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server. This member only has meaning for connections using the SSL or TLS protocols. One of the following values will be returned:

Constant	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertIssuer

A pointer to a string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertSubject

A pointer to a string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszFingerprint

A pointer to a string which contains a sequence of hexadecimal values that uniquely identify the server. This member is only used when a connection has been established using the Secure Shell (SSH) protocol.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Unicode: Implemented as Unicode and ANSI versions.

SYSTEMTIME Structure

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include windows.h.

Internet Server Class Library

A general purpose TCP/IP networking library for developing server applications.

Reference

- [Data Members](#)
- [Class Methods](#)
- [Event Handlers](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

Class Name	CInternetServer
File Name	CSWSKV10.DLL
Version	10.0.1468.2518
LibID	1437629B-0693-44DE-93ED-1482DBEFE8DC
Import Library	CSWSKV10.LIB
Dependencies	None
Standards	RFC 768, RFC 791, RFC 793

Overview

The Internet Server class library provides a simplified interface for creating event-driven, multithreaded server applications using the TCP/IP protocol. Each instance of the Internet Server class represents a server, and each active client connection is managed internally and referenced by a handle which uniquely identifies the client session. The class library supports secure connections using the standard SSL and TLS protocols and can be used to create secure, custom server programs.

This class is designed to be used as a base class from which your own server class is derived. To exchange data with the clients that connect to the server, you should override the default events such as **OnConnect** and **OnRead**. Most interaction with the clients occur within these event handlers. Because the client sessions are managed in worker threads that are separate from the main UI thread of your application, you may perform a blocking operation in response to an event without affecting the other clients that are connected to the server.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

CInternetServer Public Data Members

Member Variables	Description
m_nBacklog	The size of the backlog connection queue for the server
m_nMaxClients	The maximum number of active client sessions accepted by the server
m_nMaxClientsPerAddress	The maximum number of clients per IP address accepted by the server
m_dwOptions	The options specified when creating an instance of the server
m_nPriority	The priority specified when creating an instance of the server
m_dwStackSize	The initial size of the stack allocated for threads created by the server
m_nTimeout	The timeout period in seconds waiting for a blocking operation to complete

CInternetServer::m_nBacklog

`UINT m_nBacklog;`

The size of the backlog connection queue for the server.

Remarks

The **m_nBacklog** data member is a public variable that specifies the size of the queue allocated for pending client connections. A value of zero specifies that the queue should be set to a reasonable default value. On Windows server platforms, the maximum value is large enough to queue several hundred pending connections. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

[CInternetServer](#)

CInternetServer::m_nMaxClients

UINT m_nMaxClients;

The maximum number of clients that are permitted to connect to the server.

Remarks

The **m_nMaxClients** data member is a public variable that specifies the maximum number of clients that are permitted to establish a connection with the server. After this limit is reached, the server will reject additional connections until the number of active clients drops below this threshold. A value of zero specifies that there is no fixed limit on the active number of client connections. Changing the value of this data member does not have an effect on an active instance of the server. To change the maximum number of clients on an active server, use the **Throttle** method.

The actual number of client connections that can be accepted depends on the amount of memory available to the server process. Sockets are allocated from the non-paged memory pool, so the actual number of sockets that can be created system-wide depends on the amount of physical memory that is installed. If the server will be accessible over the Internet, it is recommended that you limit the maximum number of client connections to a reasonable value.

See Also

[CInternetServer](#), [Throttle](#)

CInternetServer::m_nMaxClientsPerAddress

UINT m_nMaxClientsPerAddress;

The maximum number of clients that are permitted to connect to the server from a single IP address.

Remarks

The **m_nMaxClientsPerAddress** data member is a public variable that specifies the maximum number of clients that are permitted to establish a connection with the server from a single IP address. After this limit is reached, the server will reject additional connections until the number of active clients drops below this threshold. A value of zero specifies that there is no limit on the active number of client connections per IP address. Changing the value of this data member does not have an effect on an active instance of the server. To change the maximum number of clients on an active server, use the **Throttle** method.

See Also

[CInternetServer](#), [Throttle](#)

CInternetServer::m_dwOptions

DWORD m_dwOptions;

The default options used when starting an instance of the server.

Remarks

The **m_dwOptions** data member is a public variable that specifies the default options that should be used when starting an instance of the server. This variable can be modified directly or by calling the **SetOptions** method. For a list of available server options, see [Server Option Constants](#). Changing the value of this data member does not have an effect on an active instance of the server.

See Also

[CInternetServer](#), [GetOptions](#), [SetOptions](#)

CInternetServer::m_nPriority

`INT m_nPriority;`

The priority specified when creating an instance of the server.

Remarks

The **m_nPriority** data member is a public variable that specifies the which specifies the priority for the server and all client sessions. Changing the value of this data member does not have an effect on an active instance of the server. It may be one of the following values:

Constant	Description
INET_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. It is recommended that most applications use this priority.
INET_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the client session. It is typically used with lightweight services running in the background that are designed for few client connections. The client thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
INET_PRIORITY_LOW	This priority lowers the overall resource utilization for the client session and meters the processor utilization for the client session. The client thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
INET_PRIORITY_HIGH	This priority increases the overall resource utilization for the client session and the thread will be given higher scheduling priority. It can be used when it is important for the client session thread to be highly responsive. It is not recommended that this priority be used on a system with a single processor.
INET_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The thread will be given higher scheduling priority and will be more responsive to the remote host. It is not recommended that this priority be used on a system with a single processor.

See Also

[CInternetServer](#), [GetPriority](#), [SetPriority](#)

CInternetServer::m_dwStackSize

```
DWORD m_dwStackSize;
```

The initial size of the stack allocated for threads created by the server.

Remarks

The **m_dwStackSize** data member is a public variable that specifies the initial amount of memory that is committed to the stack for each thread created by the server. A value of zero specifies that the default stack size should be used, which is 256K for 32-bit processes and 512K for 64-bit processes. This variable can be modified directly or by calling the **SetStackSize** method. Changing the value of this data member does not have an effect on an active instance of the server. It is recommended that most applications use the default stack size.

See Also

[CInternetServer](#), [GetStackSize](#), [SetStackSize](#)

CInternetServer::m_nTimeout

DWORD m_nTimeout;

The default options used when starting an instance of the server.

Remarks

The **m_nTimeout** data member is a public variable that specifies the number of seconds the server should wait for a client to perform a network operation. If the client does not exchange any information with the server within this period of time, a timeout event will occur. The timeout value affects all clients that are connected to the server. This variable can be modified directly or by calling the **SetTimeout** method. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

[CInternetServer](#), [GetTimeout](#), [SetTimeout](#)

CInternetServer Class Methods

Class	Description
CInternetServer	Constructor which initializes the current instance of the class
~CInternetServer	Destructor which releases resources allocated by the class
Method	Description
Abort	Abort the connection and immediately close the socket
AsyncNotify	Enable or disable asynchronous notification of changes in server status
AttachHandle	Attach the specified server handle to this instance of the class
Broadcast	Write data to all active clients currently connected to the server
Cancel	Cancel a blocking operation for the specified client session
CompareAddress	Compare two IP addresses to determine if they are identical
DetachHandle	Detach the server handle from the current instance of this class
DisableSecurity	Disable secure communication with the client
DisableTrace	Disable logging of socket function calls to the trace log
Disconnect	Disconnect the client, closing the socket handle and terminating the session
EnableSecurity	Enable secure communication with the client
EnableTrace	Enable logging of socket function calls to a file
EnumClients	Returns a list of active client connections established with the server
EnumNetworkAddresses	Return the list of network addresses that are configured for the local host
FindClient	Returns a handle to the client which matches the specified client ID or moniker
FormatAddress	Convert an IP address in binary format into a printable string
Flush	Flush the send and receive buffers for the specified client session
GetActiveClient	Return the socket handle for the active client session
GetAdapterAddress	Return the IP or MAC assigned to the specified network adapter
GetAddress	Convert an IP address string to a binary format
GetAddressFamily	Return the address family for the specified IP address
GetBacklog	Return the size of the backlog connection queue for the server
GetClientAddress	Return the IP address and port number for the specified client session
GetClientData	Returns the application defined data associated with the specified client session
GetClientHandle	Returns the handle for a specific client session based on its ID number
GetClientId	Returns the unique ID number assigned to the specified client session
GetClientIdleTime	Returns the amount of time the specified client session has been idle
GetClientMoniker	Returns the string alias associated with the specified client session
GetClientPort	Returns the remote port number used by the client to establish the connection
GetClientServer	Returns a socket handle to the server for the specified client socket
GetClientServerById	Returns a socket handle to the server for the specified session identifier
GetClientThreadId	Returns the thread ID for the specified client

GetClientThreads	Returns the number of client session threads created by the server
GetErrorString	Return a description for the specified error code
GetExternalAddress	Return the external IP address assigned to the local system
GetHandle	Return the client handle used by this instance of the class
GetHostAddress	Return the IP address assigned to the specified hostname
GetHostName	Return the hostname assigned to the specified IP address
GetLastError	Return the last error code
GetLocalAddress	Return the local IP address and port number for the server
GetLocalName	Return the hostname assigned to the local system
GetOptions	Return the current server options
GetPriority	Return the current priority assigned to the server
GetStackSize	Return the initial size of the stack allocated for threads created by the server
GetStatus	Return the current status of the server
GetStreamInfo	Return information about the current stream I/O operation
GetThreadClient	Return the handle for the client session that is being managed by the specified thread
GetTimeout	Return the timeout interval for blocking operations in seconds
IsActive	Determine if the server is currently active
IsAddressNull	Determine if the specified IP address is a null address
IsAddressRoutable	Determine if the specified IP address is routable over the Internet
IsInitialized	Determine if the class has been successfully initialized
IsListening	Determine if the server is listening for client connections
IsLocked	Determine if the server is currently in a locked state
IsProtocolAvailable	Determine if the specified protocol and address family are supported
IsReadable	Determine if data is available to be read from the client
IsWritable	Determine if data can be sent to the client without causing the thread to block
Lock	Lock the server, causing all other client threads to block until it is unlocked
MatchHostName	Match a host name against one more strings that may contain wildcards
Peek	Read data from the client without removing it from the socket buffer
Read	Read data from the client
ReadLine	Read a line of data from the client, storing it in a string buffer
ReadStream	Read a stream of data from the client and store it in the specified buffer
Reject	Reject a pending client connection
Restart	Restart the server, terminating all active client sessions
Resume	Resume accepting client connections on the specified server
SetBacklog	Set the size of the backlog connection queue for the server
SetCertificate	Specify the server certificate that should be used with secure connections
SetClientData	Associate application defined data with the specified client session
SetClientMoniker	Associate a unique string alias with the specified client session
SetLastError	Set the last error code

SetOptions	Set one or more server options
SetPriority	Set the priority assigned to the server
SetTimeout	Set the timeout interval used when waiting for a blocking operation to complete
ShowError	Display a message box with a description of the specified error
Start	Begin listening for client connections on the specified address and port
Stop	Stop listening for connections and terminate all client sessions
StoreStream	Read a stream of data from the client and store it in a file
Suspend	Suspend accepting client connections and optionally reject or disconnect clients
Throttle	Limit the number of active client connections, connections per address and connection rate
Unlock	Unlock the server, allowing other client threads to resume execution
Write	Write data to the client
WriteLine	Write a line of data to the client, terminated with a carriage-return and linefeed
WriteStream	Write a stream of data to the client

CInternetServer::CInternetServer Method

CInternetServer();

The **CInternetServer** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **InetInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[~CInternetServer](#), [IsInitialized](#)

CInternetServer::~CInternetServer

`~CInternetServer();`

The **CInternetServer** destructor releases resources allocated by the current instance of the **CInternetServer** object. It also uninitialized the library if there are no other concurrent uses of the class.

Remarks

When a **CInternetServer** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the connection are destroyed. If there are any clients connected to the server at the time the destructor is called, those client sessions will be immediately terminated.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

See Also

[CInternetServer](#)

CIInternetServer::Abort Method

```
BOOL Abort(  
    SOCKET hSocket  
);  
BOOL Abort();
```

Immediately close the socket without waiting for any remaining data to be written out.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Abort** method should only be used when the connection must be closed immediately. This method should only be used to abort client connections and should not be called within an **OnAccept** event handler. To reject an incoming client connection, use the **Reject** method.

In most cases, the server should call the **Disconnect** method to gracefully close a client connection. Aborting the connection will discard any buffered data and may cause errors or result in unpredictable behavior by the client application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[Cancel](#), [Disconnect](#), [Reject](#)

InternetServer::AsyncNotify Method

```
BOOL AsyncNotify(  
    HWND hWnd,  
    UINT uMsg  
);
```

Enable or disable asynchronous notification of changes in server status.

Parameters

hWnd

A handle to the window whose window procedure will receive the notification message.

uMsg

The user-defined message that will be sent to the notification window.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **AsyncNotify** method is used by an application to enable or disable asynchronous notifications. The message window is typically the main UI window and these notifications are used signal to the application that it should update the user interface. If the *hWnd* parameter is not NULL, it must specify a valid window handle and the user-defined message must have a value of **WM_USER** or higher. The application cannot specify a notification message that is reserved by the operating system. The pseudo-handle **HWND_BROADCAST** cannot be specified as the notification window. If the *hWnd* parameter is NULL, notifications for the specified server will be disabled.

When asynchronous notifications are enabled for a server, the server will post the user-defined message to the window whenever there is a change in status or after a client has connected or disconnected from the server. The *wParam* message parameter will contain the notification message and the *lParam* message parameter will contain the handle to the server or the client ID. The following notification messages are defined:

Constant	Description
INET_NOTIFY_STARTUP	This notification is sent when the server has started and is preparing to accept client connections. This notification is only sent once, and only if asynchronous notifications are enabled immediately after the Start method is called. This message will not be sent once the server has begun accepting client connections or when notification messages are disabled and then subsequently re-enabled at a later time. The <i>lParam</i> message parameter will specify the handle to the server.
INET_NOTIFY_LISTEN	This notification is sent when the server is listening for client connections. This notification message may be sent to the application multiple times over the lifetime of the server. If the server was suspended, this notification will be sent after the application calls the Resume

	method to resume accepting client connections. The <i>lParam</i> message parameter will specify the handle to the server.
INET_NOTIFY_SUSPEND	This notification is sent when the server suspends accepting new connections because the application has called the Suspend method. This notification message may be sent to the application multiple times over the lifetime of the server. The <i>lParam</i> message parameter will specify the handle to the server.
INET_NOTIFY_RESTART	This notification is sent when the server is restarted using the Restart method. Note that the server socket handle provided by the <i>lParam</i> message parameter will specify the new socket handle of the restarted server instance, not the original socket handle. The <i>lParam</i> message parameter will specify the handle to the server.
INET_NOTIFY_CONNECT	This notification is sent when the server accepts a client connection and the thread that manages the client session has begun processing network events for that client. This message notification will not be sent if the client connection is rejected by the server. The <i>lParam</i> message parameter will specify the unique ID of the client that connected to the server.
INET_NOTIFY_DISCONNECT	This notification is sent when the client disconnects from the server and the client socket has been closed. This notification message may not occur for each client session that is forced to terminate as the result of the server being stopped using the Stop method. The <i>lParam</i> message parameter will specify the unique ID of the client that disconnected from the server.
INET_NOTIFY_SHUTDOWN	This notification is sent when the server thread is in the process of terminating. At the time the application processes this notification message, the server handle in <i>lParam</i> will reference the defunct server and cannot be used with other server methods. The <i>lParam</i> message parameter will specify the handle to the server.

If asynchronous notifications are enabled, you should never use those notifications as a replacement for an event handler. When an event occurs, the callback function that handles the event is invoked in the context of the thread that manages the client session. The application should exchange data with the client within that event handler and not in response to a notification message. These notification messages should only be used to update the application UI in response to changes in the status of the server.

The INET_NOTIFY_CONNECT and INET_NOTIFY_DISCONNECT notifications are different from the other server notifications because the *lParam* message parameter does not specify the server handle, but rather the unique client ID associated with the session that connected to or disconnected from the server. If you need to obtain the handle to the client session using the ID, call the **GetClientHandle** method. To obtain the server handle in response to the INET_NOTIFY_CONNECT message, use the **GetClientServerById** method. Note that at the time

the application processes the INET_NOTIFY_DISCONNECT notification message, the client session will have already terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[GetClientHandle](#), [GetClientServerById](#)

CIInternetServer::AttachHandle Method

```
VOID AttachHandle(  
    SOCKET hSocket  
);
```

Attach the specified server socket handle to the current instance of the class.

Parameters

hSocket

The socket handle to the server that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a server handle created outside of the class using the SocketWrench API. Once the client handle is attached to the class, the other class member functions may be used with that server.

If a server handle already has been created for the class, that handle will be released when the new handle is attached to the class object. This will cause the server to stop and all client sessions will be terminated immediately. If you want to prevent the previous server from being stopped, you must call the **DetachHandle** method prior to attaching a new handle to the class instance.

Note that the *hSocket* parameter is presumed to be a valid server socket handle and no checks are performed to ensure that the handle references an active server. Specifying an invalid socket handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cssock10.h

Import Library: cssock10.lib

See Also

[DetachHandle](#), [GetHandle](#)

InternetServer::Broadcast Method

```
INT Broadcast(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

Sends data to all clients that are connected to the server.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server clients.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the method succeeds, the return value is the number of clients that the data was sent to. If the method fails, the return value is INET_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

The **Broadcast** method sends the contents of the buffer to all of the clients that are connected to the server. This method will block until all clients have been sent a copy of the data. There is no guarantee in which order the clients will receive and process the data that has been broadcast.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[Write](#), [WriteLine](#)

InternetServer::Cancel Method

```
BOOL Cancel(  
    SOCKET hSocket  
);
```

Cancel a blocking operation for the specified client session.

Parameters

hSocket

The handle to a client socket.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation using the same client socket handle. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Canceling a blocking operation for another client session may yield unpredictable results. If you wish to terminate the client session, it is preferable to use the **Disconnect** method rather than using this method in conjunction with the **Abort** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[Abort](#), [Disconnect](#)

CInternetServer::CompareAddress Method

```
BOOL CompareAddress(  
    LPINTERNET_ADDRESS lpAddress1,  
    LPINTERNET_ADDRESS lpAddress2  
);  
  
BOOL CompareAddress(  
    LPCTSTR lpszAddress1,  
    LPCTSTR lpszAddress2  
);
```

Compare two IP addresses to determine if they are identical.

Parameters

lpAddress1

A pointer to an INTERNET_ADDRESS structure that contains the first IP address to be compared. An alternate version of this method accepts a string that specifies the IP address to be compared.

lpAddress2

A pointer to an INTERNET_ADDRESS structure that contains the second IP address to be compared. An alternate version of this method accepts a string that specifies the IP address to be compared.

Return Value

If the method succeeds and the two addresses are identical, the return value is non-zero. If the method fails or the two addresses are not identical, the return value is zero. If either parameter is NULL, or the address family for the two addresses are not the same, the last error code will be updated. If the addresses are valid and in the same address family, but are not identical, the last error code will be set to NO_ERROR.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[GetClientAddress](#), [INTERNET_ADDRESS](#)

CInternetServer::DetachHandle Method

```
SOCKET DetachHandle();
```

The **DetachHandle** method detaches the server socket handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the server socket associated with the current instance of the class object. If the server is not active, the value `INVALID_SOCKET` will be returned.

Remarks

This method is used to detach a server handle created by the class for use with the SocketWrench API. Once the server handle is detached from the class, no other class member functions may be called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cssock10.h`

Import Library: `cswskv10.lib`

See Also

[AttachHandle](#), [GetHandle](#)

CIInternetServer::DisableSecurity Method

```
BOOL DisableSecurity(  
    SOCKET hSocket  
);  
BOOL DisableSecurity();
```

Disable secure communication with the client.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **DisableSecurity** method disables a secure session, with subsequent calls to **Read** and **Write** sending and receiving unencrypted data. It is important to note that because this method sends a shutdown message to terminate the secure session, this may cause connection to be closed by the remote host.

This method does not close the socket. Use the **Disconnect** method to close the socket and release the resources allocated for the client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[EnableSecurity](#), [SetCertificate](#)

CInternetServer::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[EnableTrace](#)

InternetServer::Disconnect Method

```
BOOL Disconnect(  
    SOCKET hSocket  
);  
  
BOOL Disconnect();
```

Disconnect the client, closing the socket handle and terminating the session.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

Once the connection has been terminated, the client socket handle is no longer valid and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new connection is established. An application should always consider the socket handle to be opaque and never depend on it being a specific value.

This method sends an internal control message that notifies the server that this session should be terminated. When the session thread is signaled that it should terminate, it will begin to release the resources allocated for that session. To ensure that the client session terminates gracefully, there may be a brief period of time where the session thread is still active after this method has returned.

The **Disconnect** method should only be used to terminate client sessions and the server handle should never be provided as the *hSocket* parameter. To stop the server, use the **Stop** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[Abort](#), [Stop](#)

CInternetServer::EnableSecurity Method

```
BOOL EnableSecurity(  
    SOCKET hSocket  
);  
BOOL EnableSecurity();
```

Enable secure communication with the client.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableSecurity** method enables a secure communications session with the remote host, automatically negotiating the encryption algorithm and validating the certificate specified by a previous call to the **SetCertificate** method. This method will cause the calling thread to block and wait for the client to initiate the TLS handshake.

This method is typically used to implement support for explicit TLS connections, where the client establishes a standard, non-secure connection to the server and then negotiates a secure connection at a later point. Usually this is done by the client sending a specific command to the server, and the server calls **EnableSecurity** from within the **OnRead** event handler that processes the command. If the method succeeds, all subsequent calls to **Read** and **Write** to receive and send data will be encrypted.

This method is only used to enable a secure connection for a specific client session. If all client connections should be secure, then call the **SetOptions** method to specify the INET_OPTION_SECURE option prior to starting the server and call the **SetCertificate** method to specify the server certificate that should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[DisableSecurity](#), [SetCertificate](#), [SetOptions](#)

CIInternetServer::EnableTrace Method

```
BOOL EnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **EnableTrace** method enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

This method will enable logging for all network function calls made by the server process, not for a particular client session or socket handle. The TRACE_HEXDUMP flag will include all of the data exchanged between the server and the clients connected to it. This has the potential to generate very large log files that can negatively impact the performance of the server. It is recommended that you only enable trace logging for debugging purposes when absolutely necessary.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: csoskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableTrace](#)

CInternetServer::EnumClients Method

```
INT EnumClients(  
    INTERNET_ADDRESS& ipAddress,  
    SOCKET * lpClients,  
    INT nMaxClients  
);  
  
INT EnumClients(  
    LPCTSTR lpszAddress,  
    SOCKET * lpClients,  
    INT nMaxClients  
);  
  
INT EnumClients(  
    SOCKET * lpClients,  
    INT nMaxClients  
);
```

Return a list of active client connections established with the server.

Parameters

ipAddress

A reference to an INTERNET_ADDRESS structure that contains the IP address that should be matched against the clients connected to the server. Only those clients that have connected to the server from this address will be returned in the *lpClients* array.

lpszAddress

A string that specifies the IP address that should be matched against the clients connected to the server. Only those clients that have connected to the server from this address will be returned in the *lpClients* array.

lpClients

Pointer to an array which will contain the socket handle for each active client session when the method returns. If this parameter is NULL, then the method will return the number of active client connections established with the server.

nMaxClients

Maximum number of socket handles to be returned in the *lpClients* array. If the *lpClients* parameter is NULL, this parameter should have a value of zero.

Return Value

If the method succeeds, the return value is the number of active client connections to the server. A return value of zero indicates that there are either no active client sessions, or no clients have connected using the specified IP address. If the method fails, the return value is FTP_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

If the *nMaxClients* parameter is less than the number of active client connections, the method will fail. To dynamically determine the number of active connections, call the method with the *lpClients* parameter with a value of NULL, and the *nMaxClients* parameter with a value of zero.

This method will not enumerate clients that have disconnected from the server, even if the session thread is still active. If the server is in the process of shutting down, this method will return zero, indicating no active client sessions, even though there may be clients that are still in the process of

disconnecting from the server. To determine the actual number of client sessions regardless of their status, use the **GetClientThreads** method.

Example

```
// Populate a listbox with the IP address for each client
pListBox->ResetContent();

INT nClients = pServer->EnumClients();
if (nClients > 0)
{
    SOCKET *phClients = new SOCKET[nClients];

    nClients = pServer->EnumClients(phClients, nClients);
    if (nClients == INET_ERROR)
    {
        // Unable to obtain list of connected clients
        return;
    }

    for (INT nIndex = 0; nIndex < nClients; nIndex++)
    {
        CString strAddress;

        if (pServer->GetClientAddress(phClients[nIndex], strAddress))
            pListBox->AddString(strAddress);
    }

    // Free the memory allocated for the socket handles
    delete phClients;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cswsock10.h

Import Library: cswskv10.lib

See Also

[GetClientAddress](#), [GetClientThreads](#)

InternetServer::EnumNetworkAddresses Method

```
INT EnumNetworkAddresses(  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddressList,  
    INT nMaxAddresses  
);  
  
INT EnumNetworkAddresses(  
    LPINTERNET_ADDRESS lpAddressList,  
    INT nMaxAddresses  
);
```

The **EnumNetworkAddresses** method returns the list of network addresses that are configured for the local host.

Parameters

nAddressFamily

An integer which identifies the type of IP address that should be returned by this function. It may be one of the following values:

Constant	Description
INET_ADDRESS_ANY	Return both IPv4 or IPv6 addresses for the local host, depending on how the system is configured and which network interfaces are enabled. This option is only recommended for applications that support IPv6.
INET_ADDRESS_IPV4	Specifies that the addresses should be in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the addresses should be in IPv6 format. All bytes in the <i>ipNumber</i> array are significant.

lpAddressList

A pointer to an array of [INTERNET_ADDRESS](#) structures that will contain the IP address of each local network interface.

nMaxAddresses

Maximum number of addresses to be returned.

Return Value

If the function succeeds, the return value is the number of network addresses that are configured for the local host. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

If the *nAddressFamily* parameter is specified as INET_ADDRESS_ANY, the application must be prepared to handle IPv6 addresses because it is possible that an IPv6 address string has been specified. For legacy applications that only recognize IPv4 addresses, the *nAddressFamily* member should always be specified as INET_ADDRESS_IPV4 to ensure that only IPv4 addresses are

returned.

If this method is called without specifying the address family, the value `INET_ADDRESS_IPV4` is used. This is provided primarily for compatibility with legacy applications and it is recommended that you explicitly specify the address family.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[FormatAddress](#), [GetHostAddress](#), [GetHostName](#)

InternetServer::FindClient Method

```
SOCKET FindClient(  
    UINT nClientId  
);  
  
SOCKET FindClient(  
    LPCTSTR lpszMoniker  
);
```

Returns a handle to the client which matches the specified client ID or moniker.

Parameters

nClientId

An unsigned integer that specifies a unique client ID for the session. This value must be greater than zero.

lpszMoniker

A pointer to a string which specifies the client moniker to search for. This parameter cannot be NULL and cannot specify an empty string.

Return Value

If the method succeeds, the return value is the handle to the client socket for the session that matches the specified client ID or moniker. If the method fails, the return value is INVALID_SOCKET. To get extended error information, call **GetLastError**.

Remarks

A client moniker is a string which can be used to uniquely identify a specific client session aside from its socket handle. A moniker can be assigned to the client session using the **SetClientMoniker** method. This method will search all active client sessions for the server, and returns the socket handle to the client that matches the specified moniker. If there is no match, an error will be returned.

The moniker can be any string value, however monikers are not case sensitive and may not contain embedded null characters. The maximum length of a moniker is 127 characters.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: csoskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientMoniker](#), [SetClientMoniker](#)

InternetServer::Flush Method

```
BOOL Flush(  
    SOCKET hSocket  
);  
BOOL Flush();
```

Flush the send and receive buffers for the specified client session.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Flush** method will flush any data waiting to be read or written to the remote host . It is important to note that this method is not similar to flushing data to a disk file; it does not ensure that a specific block of data has been written to the socket. For example, you should never call this function immediately after calling the **Write** method or prior to calling the **Disconnect** method.

An application never needs to use the **Flush** method under normal circumstances. This method is only to be used when the application needs to immediately return the socket to an inactive state with no pending data to be read or written. Calling this function may result in data loss and should only be used if you understand the implications of discarding any data which has been sent by the client.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cswsock10.h

Import Library: cswskv10.lib

See Also

[IsReadable](#), [IsWritable](#), [Read](#), [Write](#)

CInternetServer::FormatAddress Method

```
INT FormatAddress(  
    LPINTERNET_ADDRESS lpAddress,  
    LPTSTR lpszAddress,  
    INT cchAddress  
);
```

```
INT FormatAddress(  
    LPINTERNET_ADDRESS lpAddress,  
    CString& strAddress  
);
```

The **FormatAddress** method converts a numeric IP address to a printable string. The format of the string depends on whether an IPv4 or IPv6 address is specified.

Parameters

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure which specifies the numeric IP address that should be converted to a string.

lpszAddress

A pointer to the buffer that will contain the formatted IP address. This buffer should be at least 46 characters in length. This may also reference a **CString** object which will contain the formatted address when the method returns.

cchAddress

The maximum number of characters that can be copied into the address buffer.

Return Value

If the method succeeds, the return value is the length of the IP address string. If the method fails, the return value is `INET_ERROR`, meaning that the IP address could not be converted into a string. Typically this indicates that the pointer to the `INTERNET_ADDRESS` structure is invalid, or the data does not specify a valid IP address family.

Remarks

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientAddress](#), [GetLocalAddress](#), [INTERNET_ADDRESS](#)

CInternetServer::GetActiveClient Method

```
SOCKET GetActiveClient();
```

Return the socket handle for the active client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the socket handle for the active client session. If the method fails, the return value is INVALID_SOCKET. To get extended error information, call the **GetLastError** method.

Remarks

The **GetActiveClient** method returns a handle to the client socket for the active client session. The active session is determined by the session thread that is currently executing, and therefore is only meaningful within the context of a server event handler such as **OnConnect** or **OnRead**. The value returned by the his method is the same as the client socket handle that is passed to the event handler.

This method will fail within an **OnAccept** event handler because at that point the connection has not yet been accepted, therefore there is no active client session. It will also fail if called outside of an event handler. To obtain the socket handle associated with a particular session thread, use the **GetThreadClient** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: csuskv10.lib

See Also

[GetClientHandle](#), [GetClientThreadId](#), [GetThreadClient](#)

CInternetServer::GetAdapterAddress Method

```
INT GetAdapterAddress(  
    INT nAdapterIndex,  
    INT nAddressType,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

```
INT GetAdapterAddress(  
    INT nAdapterIndex,  
    INT nAddressType,  
    CString& strAddress  
);
```

Return the IP or MAC assigned to the specified network adapter.

Parameters

nAdapterIndex

An integer value that identifies the network adapter.

nAddressType

An integer value which specifies the type of address that should be returned:

Constant	Description
INET_ADAPTER_IPV4	The address string will contain the primary IPv4 unicast address assigned to the network adapter.
INET_ADAPTER_IPV6	The address string will contain the primary IPv6 unicast address assigned to the network adapter.
INET_ADAPTER_MAC	The address string will contain the media access control (MAC) address assigned to the network adapter.

lpszAddress

A string buffer that will contain the IP or MAC address assigned to the adapter. This parameter cannot be NULL and it is recommended that it be at least 64 characters in length to provide enough space for any address type. An alternate form of the method accepts a **CString** argument which will contain the address.

nMaxLength

The maximum number of characters that can be copied into the string buffer, including the terminating null character. If the buffer is too small to store the complete address, this method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied to the string buffer, not including the terminating null character. A return value of zero indicates that the requested address type has not been assigned to the adapter. If the method fails, the return value is INET_ERROR and this typically indicates that either the adapter index is invalid or the string buffer is not large enough to store the complete address. To get extended error information, call **GetLastError**.

Remarks

The **GetAdapterAddress** method will return the IPv4, IPv6 or MAC address assigned to a specific network adapter. The primary network adapter has an index value of zero, and it increments for each adapter that is configured on the local system.

The media access control (MAC) address is a 48 bit or 64 bit value that is assigned to each network interface and is used for identification and access control. All network devices on the same subnet must be assigned their own unique MAC address. Unlike IP addresses which may be assigned dynamically and can be frequently changed, MAC addresses are considered to be more permanent because they are usually assigned by the device manufacturer and stored in firmware. Note that in some cases it is possible to change the address assigned to a device, and virtual network interfaces may have configurable MAC addresses.

This method returns the MAC address string as sequence of hexadecimal values separated by a colon. An example of a 48 bit MAC address would be "01:23:45:67:89:AB". Note that some virtual network adapters may not have a MAC address assigned to them, in which case this method would return zero.

This method will ignore network adapters that have been disabled, as well as those that are bound to a virtual loopback interface. If the system has dial-up networking or virtualization software installed, this method may also return IP addresses assigned to a virtualized network adapters installed by that software.

Example

```
// Display the IPv4 address assigned to each network adapter
for (INT nIndex = 0;; nIndex++)
{
    CString strAddress;

    if (pServer->GetAdapterAddress(nIndex, INET_ADAPTER_IPV4, strAddress) ==
INET_ERROR)
        break;

    _tprintf(_T("Adapter %d: %s\n"), nIndex, szAddress);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[EnumNetworkAddresses](#), [GetLocalAddress](#), [GetLocalName](#)

CInternetServer::GetAddress Method

```
INT GetAddress(  
    LPCTSTR LpszAddress,  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS LpAddress  
);
```

```
INT GetAddress(  
    LPCTSTR LpszAddress,  
    LPINTERNET_ADDRESS LpAddress  
);
```

The **GetAddress** method converts an IP address string to binary format.

Parameters

LpszAddress

A pointer to a null terminated string which specifies an IP address. This method recognizes the format for both IPv4 and IPv6 format addresses.

nAddressFamily

An integer which identifies the type of IP address specified by the *LpszAddress* parameter. It may be one of the following values:

Constant	Description
INET_ADDRESS_UNKNOWN	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on the value of the <i>LpszAddress</i> parameter.
INET_ADDRESS_IPV4	Specifies that the address should be in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero. If the <i>LpszAddress</i> parameter does not specify a valid IPv4 address string, this method will fail.
INET_ADDRESS_IPV6	Specifies that the address should be in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. If the <i>LpszAddress</i> parameter does not specify a valid IPv6 address string, this method will fail.

LpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the IP address.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is [INET_ERROR](#). To get extended error information, call **GetLastError**.

Remarks

If the *nAddressFamily* parameter is specified as [INET_ADDRESS_UNKNOWN](#), the application must be prepared to handle IPv6 addresses because it is possible that an IPv6 address string has been specified. For legacy applications that only recognize IPv4 addresses, the *nAddressFamily* member

should always be specified as `INET_ADDRESS_IPV4` to ensure that only IPv4 addresses are returned and any attempt to specify an IPv6 address string would result in an error.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **IsProtocolAvailable** method. If an IPv6 stack is not installed, this method will fail if the *lpszAddress* parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FormatAddress](#), [IsAddressNull](#), [IsAddressRoutable](#), [IsProtocolAvailable](#), [INTERNET_ADDRESS](#)

CInternetServer::GetAddressFamily Method

```
INT GetAddressFamily(  
    LPCTSTR lpszAddress,  
);
```

Return the address family for the specified IP address.

Parameters

lpszAddress

A pointer to a string which specifies an IP address. This method recognizes the format for both IPv4 and IPv6 format addresses.

Return Value

If the method succeeds, the return value is the address family for the specified IP address and may be one of the values listed below. If the method fails, the return value is INET_ADDRESS_UNKNOWN. To get extended error information, call the **GetLastError** method.

Constant	Description
INET_ADDRESS_IPV4	The address passed to the method is a valid IPv4 address.
INET_ADDRESS_IPV6	The address passed to the method is a valid IPv6 address.

Remarks

The **GetAddressFamily** method returns the address family associated with the specified IP address string. This can be used to determine if a string specifies a valid IPv4 or IPv6 address that can be passed to other methods such as **Connect**. Note that this method will not attempt to resolve hostnames, it will only accept IP addresses.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **IsProtocolAvailable** method. If an IPv6 stack is not installed, this method will fail if the *lpszAddress* parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IsAddressNull](#), [IsAddressRoutable](#), [IsProtocolAvailable](#), [INTERNET_ADDRESS](#)

CInternetServer::GetBacklog Method

```
UINT GetBacklog();
```

Return the size of the backlog connection queue for the server.

Parameters

None.

Return Value

The return value is the size of the queue used to accept client connections.

Remarks

The **GetBacklog** method returns the size of the queue allocated for pending client connections. A value of zero specifies that the size of the queue should be set to a reasonable default value. On Windows server platforms, the maximum value is large enough to queue several hundred pending connections. To change the size of the backlog queue, use the **SetBacklog** method prior to starting the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[SetBacklog](#), [Start](#), [Data Members](#)

CIInternetServer::GetClientAddress Method

```
INT GetClientAddress(  
    SOCKET hSocket,  
    LPINTERNET_ADDRESS lpAddress,  
    UINT * lpnRemotePort  
);  
  
INT GetClientAddress(  
    SOCKET hSocket,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);  
  
INT GetClientAddress(  
    SOCKET hSocket,  
    CString& strAddress  
);
```

Return the IP address and port number for the specified client session.

Parameters

hSocket

An optional parameter that specifies the handle to a server or client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the IP address of the client that is connected to the server. This parameter may be NULL, in which case the IP address will not be returned to the caller.

lpnRemotePort

A pointer to an unsigned integer that will contain the port number of the client that is connected to the server. This parameter may be NULL, in which case the port number will not be returned to the caller.

lpszAddress

A pointer to a string buffer that will contain the formatted IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length.

nMaxLength

The maximum number of characters that can be copied into the address buffer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

If this method is called within an **OnAccept** event handler, passing the server handle as the *hSocket* parameter will return the IP address and port number for the client that is attempting to establish the connection. If the client address is unavailable, the *ipFamily* member of the `INTERNET_ADDRESS` structure will be zero.

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

See Also

[GetClientPort](#), [OnAccept](#), [OnConnect](#), [INTERNET_ADDRESS](#)

CIInternetServer::GetClientData Method

```
BOOL GetClientData(  
    SOCKET hSocket,  
    LPVOID * LppvData  
);  
  
BOOL GetClientData(  
    LPVOID * LppvData  
);
```

Returns the application defined data associated with the specified client session.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lppvData

A pointer to a void pointer which will contain an application defined value associated with the client session.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates that application defined data for the client session could not be retrieved. To get extended error information, call **GetLastError**.

Remarks

The **GetClientData** method is used to retrieve the application defined data that was previously associated with a client session using the **SetClientData** method. This is typically used to associate a pointer to a data structure or a class instance with a specific client handle.

This method can only be used with client socket handles created using the server interface. If the socket handle is invalid, or does not reference a client socket handle created by the server, the *lppvData* pointer passed to this method will be initialized to a value of NULL and the method will return a value of zero.

If this method is called with a valid socket handle and there is no data associated with the socket, the method will return a non-zero value and the *lppvData* pointer will be returned with a NULL value. Before dereferencing the pointer returned by this method, the application should always check the return value to ensure the method succeeded and make sure that the pointer is not NULL.

Example

```
UINT *pnValue1 = new UINT;  
UINT *pnValue2 = NULL;  
  
*pnValue1 = 1234;  
  
if (!pServer->SetClientData(hClient, pnValue1))  
{  
    // Unable to associate the data with this session  
    return;
```

```
}  
  
if (!pServer->GetClientData(hClient, (LPVOID *)&pnValue2))  
{  
    // Unable to retrieve the data associated with this session  
    return;  
}  
  
// pnValue2 == pnValue1  
printf("The value of the user defined data is %u\n", *pnValue2);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[SetClientData](#)

CIInternetServer::GetClientHandle Method

```
SOCKET GetClientHandle(  
    UINT nClientId  
);
```

Returns the handle for a specific client session based on its ID number.

Parameters

nClientId

An unsigned integer value which uniquely identifies the client session.

Return Value

If the method succeeds, the return value is the socket handle for the specified client session. If the method fails, the return value is INVALID_SOCKET. To get extended error information, call **GetLastError**.

Remarks

Each client connection that is accepted by the server is assigned a unique numeric value called the client ID. The **GetClientHandle** method will return the socket handle that is associated with the specified client ID. Unlike socket handles, which are reused by the operating system, the client ID is guaranteed to be unique throughout the lifetime of the server. To obtain the ID associated with the client session, use the **GetClientId** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[GetClientId](#), [GetClientMoniker](#), [SetClientMoniker](#), [GetThreadClient](#)

CIInternetServer::GetClientId Method

```
UINT GetClientId(  
    SOCKET hSocket  
);  
UINT GetClientId();
```

Returns the unique ID number assigned to the specified client session.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is an unsigned integer value which uniquely identifies the client session. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

Each client connection that is accepted by the server is assigned a unique numeric value. This value can be obtained by calling the **GetClientId** method and used by the application to identify that client session. The **GetClientHandle** method can then be used to obtain the client socket handle for the session, based on that client ID. It is important to note that the actual value of the client ID should be considered opaque. It is only guaranteed that the value will be greater than zero, and that it will be unique to the client session.

While it is possible for a client socket handle to be reused by the operating system, client IDs are unique throughout the life of the server session and are never duplicated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[GetClientHandle](#), [GetClientMoniker](#), [SetClientMoniker](#)

InternetServer::GetClientIdleTime Method

```
DWORD GetClientIdleTime(  
    SOCKET hSocket  
);  
  
DWORD GetClientIdleTime();
```

Returns the number of milliseconds that the specified client session has been idle.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is an unsigned integer value which specifies the number of milliseconds the client session has been idle. If the method fails, the return value is INFINITE. To get extended error information, call **GetLastError**.

Remarks

The **GetClientIdleTime** method will return the number of milliseconds that have elapsed since data was exchanged with the client. The elapsed time is limited to the resolution of the system timer, which is typically in the range of 10 milliseconds to 16 milliseconds.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[GetClientHandle](#), [GetClientId](#), [GetClientMoniker](#), [GetTimeout](#)

CInternetServer::GetClientMoniker Method

```
INT GetClientMoniker(  
    SOCKET hClient,  
    LPTSTR lpszMoniker,  
    INT nMaxLength  
);
```

```
INT GetClientMoniker(  
    SOCKET hClient,  
    CString& strMoniker  
);
```

The **GetClientMoniker** method returns the moniker associated with the specified client session.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpszMoniker

Pointer to a string buffer that will contain the moniker for the specified client session when the method returns. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

The maximum number of characters that may be copied into the string buffer. The buffer must be large enough to store the moniker and a terminating null character. The maximum length of a moniker is 127 characters.

Return Value

If the method succeeds, the return value is the number of characters in the moniker string. A return value of zero specifies that no moniker was assigned to the socket. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

A client moniker is a string which can be used to uniquely identify a specific client session aside from its socket handle. A moniker can be assigned to the client session using the **SetClientMoniker** method. This method will return the moniker that was previously assigned to the client, if any. To obtain the socket handle associated with a given moniker, use the **FindClient** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

See Also

[FindClient](#), [GetClientId](#), [SetClientMoniker](#)

InternetServer::GetClientPort Method

```
INT GetClientPort(  
    SOCKET hSocket  
);  
  
INT GetClientPort();
```

Returns the remote port number used by the client to establish the connection.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is the port number. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetClientPort** method returns the remote port number that the client is bound to. Note that this is not the port number that the server is using to listen for connections, it is the port number that the client is bound to on the remote host. Typically this is an ephemeral port, either in the range of 1025 through 5000, or greater than 32768, depending on the client operating system.

If this method is called within the **OnAccept** event handler, providing the server socket handle as the *hSocket* parameter will return the port number of the client that is attempting to establish the connection.

It is not recommended that you use the client port number for anything other than informational and logging purposes. Do not make any assumptions about the specific port number or range of port numbers that a client is using when establishing a connection to the server. The ephemeral port number that a client is bound to can vary based on the client operating system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[GetClientAddress](#), [OnAccept](#), [OnConnect](#)

CIInternetServer::GetClientServer Method

```
SOCKET HttpGetClientServer(  
    UINT nClientId  
);
```

The **GetClientServer** method returns a handle to the server that created the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the method succeeds, the return value is the handle to the server that created the client session. If the method fails, the return value is INVALID_SOCKET. To get extended error information, call the **GetLastError** method.

Remarks

The **GetClientServer** method returns the handle to the server that created the client session and is typically used within a notification message handler. If the server is in the process of shutting down, or the client session thread is terminating, this method will fail and return INVALID_SOCKET indicating that the session ID is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[AsyncNotify](#)

InternetServer::GetClientServerById Method

```
SOCKET GetClientServerById(  
    UINT nClientId  
);
```

The **GetClientServerById** method returns a socket handle to the server for the specified client session identifier.

Parameters

nClientId

Client session identifier.

Return Value

If the method succeeds, the return value is the handle to the server that created the client session. If the method fails, the return value is INVALID_SOCKET. To get extended error information, call the **GetLastError** method.

Remarks

The **GetClientServerById** method returns the handle to the server that created the client session using the client's unique identifier. The **GetClientServer** method can be used to obtain the server handle using the client socket handle rather than the client session ID. This method is typically used in conjunction with the INET_NOTIFY_CONNECT notification message to obtain the handle to the server that generated the event using the client ID passed in the *wParam* message parameter.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[AsyncNotify](#), [GetClientHandle](#), [GetClientId](#), [GetClientServer](#)

CInternetServer::GetClientThreadId Method

```
DWORD GetClientThreadId(  
    SOCKET hSocket  
);  
  
DWORD GetClientThreadId();
```

Returns the thread ID associated with the specified client.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is a thread ID. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **GetClientThreadId** method returns a thread ID that can be used to identify the thread that is managing the client session. The thread ID can be used with other Windows API functions such as **OpenThread**. Exercise caution when using thread-related functions, interfering with the normal operation of the thread can have unexpected results. You should never use this method to obtain a thread handle and then call the **TerminateThread** function to terminate a client session. This will prevent the thread from releasing the resources that were allocated for the session and can leave the server in an unstable state. To terminate a client session, use the **Disconnect** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cswsock10.h

Import Library: cswskv10.lib

See Also

[EnumClients](#), [GetActiveClient](#)

CIInternetServer::GetClientThreads Method

```
INT GetClientThreads();
```

Returns the number of client session threads created by the server.

Parameters

None.

Return Value

If the method succeeds, the return value is the number of client session threads that have been created by the server. If the method fails, the return value is INET_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

The **InetGetClientThreads** function returns the number of threads that are managing client sessions for the specified server. If there are no clients connected to the server, this function will return a value of zero. Because this function returns the number of session threads, the value returned will include those clients that are in the process of disconnecting from the server but their session thread has not yet terminated. This differs from the **InetEnumServerClients** function which will only enumerate active clients.

If you wish to determine when the last client has disconnected from the server, call this function within an event handler for the INET_EVENT_DISCONNECT event. If the function returns a value greater than one, then there are other client sessions that are either connected or in the process of terminating.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: csoskv10.lib

See Also

[EnumClients](#)

CInternetServer::GetErrorString Method

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);
```

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    CString& strDescription  
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLastError](#), [SetLastError](#), [ShowError](#)

InternetServer::GetExternalAddress Method

```
INT GetExternalAddress(  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddress,  
    INT nMaxLength  
);  
  
INT GetExternalAddress(  
    INT nAddressFamily,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);  
  
INT GetExternalAddress(  
    INT nAddressFamily,  
    CString& strAddress  
);
```

The **GetExternalAddress** method returns the external IP address for the local system.

Parameters

nAddressFamily

An integer which identifies the type of IP address that should be returned by this function. It may be one of the following values:

Constant	Description
INET_ADDRESS_IPV4	Specifies that the address should be in IPv4 format. The method will attempt to determine the external IP address using an IPv4 network connection.
INET_ADDRESS_IPV6	Specifies that the address should be in IPv6 format. The method will attempt to determine the external IP address using an IPv6 network connection and requires that the local host have an IPv6 network interface installed and enabled.

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the external IP address of the local host in binary form.

lpszAddress

A pointer to a string buffer that will contain the external IP address of the local host.

nMaxLength

The maximum length of the string that will contain the IP address when the method returns.

Return Value

In the first form of the method, if it succeeds, the return value is the IP address of the local system in numeric form. If the method fails, the return value is INET_ADDRESS_NONE. In the second form, the return value is the length of the IP address string and an error is indicated by the return value INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetExternalAddress** method returns the IP address assigned to the router that connects the

local host to the Internet. This is typically used by an application executing on a system in a local network that uses a router which performs Network Address Translation (NAT). In that network configuration, the **GetLocalAddress** method will only return the IP address for the local system on the LAN side of the network unless a connection has already been established to a remote host. The **GetExternalAddress** function can be used to determine the IP address assigned to the router on the Internet side of the connection and can be particularly useful for servers running on a system behind a NAT router.

This method requires that you have an active connection to the Internet and calling this function on a system that uses dial-up networking may cause the operating system to automatically connect to the Internet service provider. An application should always check the return value in case there is an error; never assume that the return value is always a valid address. The function may be unable to determine the external IP address for the local host for a number of reasons, particularly if the system is behind a firewall or uses a proxy server that restricts access to external sites on the Internet. If the function is able to obtain a valid external address for the local host, that address will be cached by the library for sixty minutes. Because dial-up connections typically have different IP addresses assigned to them each time the system is connected to the Internet, it is recommended that this function only be used with broadband connections where a NAT router is being used.

Calling this function may cause the current thread to block until the external IP address can be resolved and should never be used in conjunction with asynchronous socket connections. If you need to call this function in an application which uses asynchronous sockets, it is recommended that you create a new thread and call this function from within that thread.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientAddress](#), [GetHostAddress](#), [GetLocalAddress](#)

CInternetServer::GetHandle Method

```
SOCKET GetHandle();
```

The **GetHandle** method returns the socket handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the socket handle associated with the current instance of the class object. If there is no active connection, the value `INVALID_SOCKET` will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketWrench API.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[AttachHandle](#), [DetachHandle](#), [IsInitialized](#)

CInternetServer::GetHostAddress Method

```
INT GetHostAddress(  
    LPCTSTR lpszHostName,  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddress  
);
```

```
INT GetHostAddress(  
    LPCTSTR lpszHostName,  
    LPINTERNET_ADDRESS lpAddress  
);
```

The **InetGetHostAddress** method resolves the specified host name into an IP address in binary format.

Parameters

lpszHostName

A pointer to the name of the host to resolve; this may be a fully-qualified domain name or an IP address. This method recognizes the format for both IPv4 and IPv6 format addresses.

nAddressFamily

An integer which identifies the type of IP address to return. It may be one of the following values:

Constant	Description
INET_ADDRESS_UNKNOWN	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on how the host name can be resolved. By default, a preference will be given for returning an IPv4 address. However, if the host only has an IPv6 address, that value will be returned.
INET_ADDRESS_IPV4	Specifies that the address should be returned in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address should be returned in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the IP address of the specified host.

Return Value

If the method succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

This method can also be used to convert an address in dot notation to a binary format. If the method must perform a DNS lookup to resolve the hostname, the calling thread will block. To ensure future compatibility with IPv6 networks, it is important that the application does not make any assumptions about the format of the address. If the function returns successfully, the *ipFamily* member of the **INTERNET_ADDRESS** structure should always be checked to determine the type of address.

The *nAddressFamily* parameter is used to specify a preference for the type of address returned, however it is possible that a host may not have an IPv4 or IPv6 address record, in which case this function will fail. Although IPv4 is still the most common address used at this time, an application should not assume that because a given host name does not have an IPv4 address, that the host name is invalid.

If the *nAddressFamily* parameter is specified as `INET_ADDRESS_UNKNOWN`, the application must be prepared to handle IPv6 addresses because it is possible for a host name to have an IPv6 address assigned to it and no IPv4 address. For legacy applications that only recognize IPv4 addresses, the *nAddressFamily* member should always be specified as `INET_ADDRESS_IPV4` to ensure that only IPv4 addresses are returned.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **IsProtocolAvailable** method. If an IPv6 stack is not installed, this method will fail if the *lpszHostName* parameter specifies a host that only has an IPv6 (AAAA) DNS record.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientAddress](#), [GetHostName](#), [GetLocalAddress](#), [GetLocalName](#), [IsProtocolAvailable](#), [INTERNET_ADDRESS](#)

CInternetServer::GetHostName Method

```
INT GetHostName(  
    LPINTERNET_ADDRESS LpAddress,  
    LPTSTR LpszHostName,  
    INT cchHostName  
);
```

```
INT GetHostName(  
    LPINTERNET_ADDRESS LpAddress,  
    CString& strHostName  
);
```

The **GetHostName** method performs a reverse lookup, returning the host name associated with a given IP address.

Parameters

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure which specifies the IP address that should be resolved into a host name.

lpszHostName

A pointer to the buffer that will contain the host name. It is recommended that this buffer be at least 256 characters in length to accommodate the longest possible fully qualified domain name. This parameter cannot be NULL. An alternate form of the method accepts a **CString** argument which will contain the hostname.

cchHostName

The maximum number of characters that can be copied into the buffer.

Return Value

If the method succeeds, the return value is the length of the hostname. If the method fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

If the method must perform a reverse DNS lookup to resolve the IP address into a host name, the calling thread will block. This method requires that the host have a PTR record, otherwise it will fail. Because many hosts do not have a PTR record, calling this method frequently may have a negative impact on the overall performance of the application.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **IsProtocolAvailable** method. If an IPv6 stack is not installed, this method will fail if the *lpAddress* parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientAddress](#), [GetHostAddress](#), [GetLocalAddress](#), [GetLocalName](#), [IsProtocolAvailable](#), [INTERNET_ADDRESS](#)

CInternetServer::GetLastError Method

```
DWORD GetLastError();  
  
DWORD GetLastError(  
    CString& strDescription  
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError(0)** when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SOCKET or INET_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[GetErrorString](#), [SetLastError](#), [ShowError](#)

InternetServer::GetLocalAddress Method

```
INT GetLocalAddress(  
    LPINTERNET_ADDRESS LpAddress,  
    UINT * LpnPort  
);
```

```
INT GetLocalAddress(  
    LPTSTR LpszAddress,  
    INT nMaxLength  
);
```

```
INT GetLocalAddress(  
    CString& strAddress  
    UINT * LpnPort  
);
```

Return the local IP address and port number for the server.

Parameters

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the IP address of the local host. If the server is not active, this function will attempt to determine the IP address of the local host assigned by the system. If the address is not required, this parameter may be NULL.

lpszAddress

A pointer to a null terminated string that will contain the IP address of the local host. If this version of the method is used, the IP address is converted to a string format using the **FormatAddress** method. The string should be able to store at least 46 characters to ensure that both IPv4 and IPv6 formatted addresses can be returned without the possibility of a buffer overrun. An alternate form of the method accepts a **CString** argument which will contain the line of text returned by the server.

lpnPort

A pointer to an unsigned integer that will contain the local port number. If the server is active, this parameter will be set to the local port that the listening socket was bound to. If the server is not active, this parameter is ignored. If the port number is not required, this parameter may be NULL.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

To ensure future compatibility with IPv6 networks, it is important that the application does not make any assumptions about the format of the address. If the function returns successfully, the *ipFamily* member of the **INTERNET_ADDRESS** structure should always be checked to determine the type of address.

If the system is connected to the Internet through a local network and/or uses a router that performs Network Address Translation (NAT), the **GetLocalAddress** method will return the local, non-routable IP address assigned to the local system. To determine the public IP address has been assigned to the system, you should use the **GetExternalAddress** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientAddress](#), [GetExternalAddress](#), [GetHostAddress](#), [GetHostName](#), [GetLocalName](#),
[INTERNET_ADDRESS](#)

CInternetServer::GetLocalName Method

```
INT GetLocalName(  
    LPTSTR lpszHostName,  
    INT cchHostName  
);  
  
INT GetLocalName(  
    CString& strHostName  
);
```

The **GetLocalName** method returns the hostname assigned to the local system.

Parameters

lpszHostName

A pointer to the buffer that will contain the hostname. This parameter cannot be NULL. An alternate form of the method accepts a **CString** argument which will contain the local hostname.

cchHostName

The maximum number of characters that can be copied into the address buffer.

Return Value

If the method succeeds, the return value is the length of the hostname. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientAddress](#), [GetLocalAddress](#)

CInternetServer::GetOptions Method

```
DWORD GetOptions();
```

Return the current server options.

Parameters

None.

Return Value

This method returns an unsigned integer value that specifies the server options that are currently enabled for the class instance. For a list of the available options, see [Server Option Constants](#). This method returns the value of the **m_dwOptions** class member variable.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[SetOptions](#), [Data Members](#)

CInternetServer::GetPriority Method

```
INT GetPriority();
```

Return the current priority assigned to the specified server.

Parameters

None.

Return Value

If the method succeeds, the return value is the priority for the specified server. If the method fails, the return value is INET_PRIORITY_INVALID. To get extended error information, call the **GetLastError** method.

Remarks

The **GetPriority** method can be used to determine the current priority assigned to the server. It will return one of the following values:

Constant	Description
INET_PRIORITY_BACKGROUND (0)	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
INET_PRIORITY_LOW (1)	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
INET_PRIORITY_NORMAL (2)	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
INET_PRIORITY_HIGH (3)	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
INET_PRIORITY_CRITICAL (4)	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[SetPriority](#)

CInternetServer::GetStackSize Method

```
DWORD GetStackSize();
```

Return the initial size of the stack allocated for threads created by the server.

Parameters

None.

Return Value

If the method succeeds, the return value is the amount of memory that will be allocated for the stack in bytes. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetStackSize** method returns the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: csuskv10.lib

See Also

[SetStackSize](#), [Start](#)

CInternetServer::GetStatus Method

INT GetStatus();

Return the current status of the server.

Parameters

None.

Return Value

An integer value that specifies the server status.

Remarks

The return value is one of the following values:

Constant	Description
INET_SERVER_INACTIVE	The server is currently inactive.
INET_SERVER_STARTED	The server has initialized and is preparing to listen for client connections.
INET_SERVER_LISTENING	The server is actively listening for incoming client connections.
INET_SERVER_SUSPENDED	The server has been suspended and is no longer accepting client connections.
INET_SERVER_SHUTDOWN	The server has been stopped and is in the process of terminating all active client connections.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cswsock10.h

Import Library: csWSKV10.lib

See Also

[IsInitialized](#), [IsListening](#), [IsLocked](#)

CInternetServer::GetStreamInfo Method

```
BOOL GetStreamInfo(  
    SOCKET hSocket  
    LPINETSTREAMINFO lpStreamInfo  
);  
  
BOOL GetStreamInfo(  
    LPINETSTREAMINFO lpStreamInfo  
);
```

The **GetStreamInfo** function fills a structure with information about the current stream I/O operation.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpSecurityInfo

A pointer to an **INETSTREAMINFO** structure which contains information about the status of the current operation.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetStreamInfo** method returns information about the current streaming socket operation, including the average number of bytes transferred per second and the estimated amount of time until the operation completes. If there is no operation currently in progress, this method will return the status of the last successful streaming read or write performed by the client.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[ReadStream](#), [StoreStream](#), [WriteStream](#), [INETSTREAMINFO](#)

CInternetServer::GetThreadClient Method

```
SOCKET WINAPI InetGetThreadClient(  
    DWORD dwThreadId  
);
```

The **GetThreadClient** method returns the socket handle for the client session that is being managed by the specified thread.

Parameters

dwThreadId

An unsigned integer value which identifies the thread managing the client session. If this parameter has a value of zero, then the client handle for the current thread is returned.

Return Value

If the method succeeds, the return value is the socket handle for the specified client session. If the method fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

The **GetThreadClient** is used to obtain the socket handle for the client session that is being managed by the specified thread. If the specified thread ID is zero, then the method will return the client socket for the current thread, otherwise it will search the internal table of all active client sessions and return the handle to the session that is being managed by that thread.

This method will fail if the thread ID does not specify an active client session thread.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[GetActiveClient](#), [GetClientHandle](#), [GetClientId](#), [GetClientThreadId](#)

CInternetServer::GetTimeout Method

```
INT GetTimeout();
```

Return the timeout interval for blocking network operations in seconds.

Parameters

None.

Return Value

The return value is the timeout period in seconds. If there is no active server, this will return the timeout period that will be used when the server is started. A value of zero specifies that a reasonable default timeout period will be automatically selected.

Remarks

The **GetTimeout** method returns the number of seconds the server will wait for a blocking network operation to complete, such as sending or receiving data. This value also determines the amount of time that the server will wait for the client to send data before invoking the **OnTimeout** event handler for that session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cssock10.lib`

See Also

[IsReadable](#), [IsWritable](#), [SetTimeout](#), [OnTimeout](#)

CInternetServer::IsActive Method

```
BOOL IsActive();
```

Determine if the server has been started.

Return Value

This method returns a non-zero value if the server has been started. If the server is stopped this method will return zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cswsock10.h

Import Library: cswskv10.lib

See Also

[CInternetServer](#), [IsListening](#), [Start](#), [Stop](#)

CIInternetServer::IsAddressNull Method

```
BOOL IsAddressNull(  
    LPCTSTR lpszAddress  
);  
  
BOOL IsAddressNull(  
    LPINTERNET_ADDRESS lpAddress  
);
```

The **IsAddressNull** method determines if the IP address is null.

Parameters

lpszAddress

A string that specifies the IP address.

lpAddress

A pointer to an INTERNET_ADDRESS structure that specifies the IP address.

Return Value

If the method succeeds and the IP address is null, or the parameter is a NULL pointer, the return value is non-zero. If the method fails or the address is not null, the return value is zero. If the address family is not supported, the last error code will be updated. If the address is valid but not null, the last error code will be set to NO_ERROR.

Remarks

A null IP address is one where all bits for the address (32 bits for IPv4 or 128 bits for IPv6) are zero. This is a special address that is typically used when creating a passive socket that should listen for connections on all available network interfaces.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[GetAddress](#), [IsAddressRoutable](#), [INTERNET_ADDRESS](#)

InternetServer::IsAddressRoutable Method

```
BOOL IsAddressRoutable(  
    LPCTSTR lpszAddress  
);  
  
BOOL IsAddressRoutable(  
    LPINTERNET_ADDRESS lpAddress  
);
```

The **IsAddressRoutable** method determines if the IP address is routable over the Internet.

Parameters

lpszAddress

A string that specifies the IP address.

lpAddress

A pointer to an INTERNET_ADDRESS structure that specifies the IP address.

Return Value

If the method succeeds and the IP address is routable over the Internet, the return value is non-zero. If the method fails or the address is not routable, the return value is zero. If the parameter is NULL, or the address family is not supported, the last error code will be updated. If the address is valid but not routable, the last error code will be set to NO_ERROR.

Remarks

A routable IP address is one that can be reached by anyone over the public Internet. These are also commonly referred to as "public addresses" which are typically assigned to networks and individual hosts by an Internet service provider. There are also certain addresses that are not routable over the Internet, and used to address systems over a local network or private intranet. This function can be used to determine if a given IP address is public (routable) or private.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[GetAddress](#), [GetExternalAddress](#), [IsAddressNull](#), [INTERNET_ADDRESS](#)

CInternetServer::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[CInternetServer](#), [IsListening](#), [IsLocked](#)

CInternetServer::IsListening Method

```
BOOL IsListening();
```

Determine if the server is listening for client connections.

Parameters

None.

Return Value

If the server has started and is listening for client connections, the method returns a non-zero value. If the server is not listening for connections, the return value is zero.

Remarks

The **IsListening** method determines if the server has been started and is actively listening for incoming connection requests from client applications. This method will return zero if the server is not active, if it has been suspended using the **Suspend** method or if the **Stop** method has been called to shutdown the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[IsActive](#), [Start](#), [Stop](#), [Suspend](#)

CInternetServer::IsLocked Method

BOOL IsLocked();

Determine if the server is currently in a locked state.

Parameters

None.

Return Value

If the server is locked, the method returns a non-zero value. If the server is not locked, the return value is zero.

Remarks

The **IsLocked** method determines if a server has been locked using the **Lock** method. Only the thread that has locked the server may interact with it and all other threads will block when they attempt to perform a network operation. After the server is unlocked, the blocked threads will resume normal execution. If the application has created multiple instances of the **CInternetServer** class, this method will return a non-zero value if any of those servers have been locked, not only the current instance.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[Lock](#), [Unlock](#)

CInternetServer::IsProtocolAvailable Method

```
BOOL IsProtocolAvailable(  
    INT nAddressFamily,  
    INT nProtocol  
);
```

The **IsProtocolAvailable** method determines if the operating system supports creating a socket for the specified address family and protocol.

Parameters

nAddressFamily

An integer which identifies the address family that should be checked. It should be one of the following values:

Constant	Description
INET_ADDRESS_IPV4	Specifies that the function should determine if it can create an Internet Protocol version 4 (IPv4) socket. This requires that the system have an IPv4 TCP/IP stack bound to at least one network adapter on the local system. All Windows systems include support for IPv4 by default.
INET_ADDRESS_IPV6	Specifies that the function should determine if it can create an Internet Protocol version 6 (IPv6) socket. This requires that the system have an IPv6 TCP/IP stack bound to at least one network adapter on the local system. Windows XP and Windows Server 2003 includes support for IPv6, however it is not installed by default. Windows Vista and later versions include support for IPv6 and enable it by default.

nProtocol

An integer which identifies the protocol that should be checked. It should be one of the following values:

Constant	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This requires that the system be capable of creating a stream socket using the specified address family.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. This requires that the system be capable of creating a datagram socket using the specified address family.

Return Value

If the the system is capable of creating a socket using the specified address family and protocol, this method will return a non-zero value. If the combination of address family and protocol is not supported, this method will return a value of zero.

Remarks

The **IsProtocolAvailable** method is used to determine if the operating system supports creating a particular type of socket. Typically it is used by an application to determine if the system has an IPv6 TCP/IP stack installed and configured. By default, all Windows systems will have an IPv4 stack installed if the system has a network adapter. However, not all systems may have an IPv6 stack installed, particularly older Windows XP and Windows Server 2003 systems. Note that if an IPv6 stack is not installed, the library will not recognize IPv6 addresses and cannot resolve host names that only have an IPv6 (AAAA) record, even if the address or host name is valid.

Example

```
if (!pSocket->IsProtocolAvailable(INET_ADDRESS_IPV6, INET_PROTOCOL_TCP))
{
    AfxMessageBox(_T("This system does not support IPv6"), MB_ICONEXCLAMATION);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[GetAddress](#), [GetHostAddress](#), [GetHostName](#)

CInternetServer::IsReadable Method

```
BOOL IsReadable();  
  
BOOL IsReadable(  
    SOCKET hSocket  
);  
  
BOOL IsReadable(  
    SOCKET hSocket,  
    INT nTimeout,  
    LPDWORD lpdwAvail  
);
```

The **IsReadable** method is used to determine if data is available to be read from the client.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

nTimeout

Timeout for remote host response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the server can read data from the client within the specified timeout period, the method returns a non-zero value. If there is no data available to be read, the method returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

If the connection is secure, the value returned in *lpdwAvail* will reflect the number of bytes available in the encrypted data stream. The actual amount of data available to the application after it has been decrypted will vary.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[IsWritable](#), [Peek](#), [Read](#), [ReadLine](#), [ReadStream](#)

CIInternetServer::IsWritable Method

```
BOOL IsWritable(  
    INT nTimeout  
);
```

The **IsWritable** method is used to determine if data can be written to the remote host.

Parameters

nTimeout

Timeout for remote host response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If data can be sent to the client within the specified timeout period, the method returns a non-zero value. The method will return zero if the socket send buffer is full.

Remarks

The **IsWritable** method cannot be used to determine the amount of data that can be sent to the client without blocking the current thread. A non-zero return value only indicates that the send buffer is not full and can accept some data. In most cases, it is recommended that larger blocks of data be broken into smaller logical blocks rather than attempting to send it all of the data at once. For very large streams of data, it is recommended that you use the **WriteStream** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: csuskv10.lib

See Also

[IsReadable](#), [Write](#), [WriteLine](#), [WriteStream](#)

InternetServer::Lock Method

```
BOOL Lock();
```

Lock the server, causing other client threads to block until it is unlocked.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **Lock** method causes the specified server to enter a locked state where only the current thread may interact with the server and the clients that are connected to it. While a server is locked, all other threads will block when they attempt to perform a network operation. When the server is unlocked, the blocked threads will resume normal execution.

This method should be used carefully, and a server should never be left in a locked state for an extended period of time. It is meant to be used when the server process updates a global data structure and it must prevent any other threads from performing a network operation during the update. Only one server can be locked at any one time, and once a server has been locked, it can only be unlocked by the same thread.

The program should always check the return value from this method, and should never assume that the lock has been established. If more than one thread attempts to lock a server at the same time, there is no guarantee as to which thread will actually establish the lock. If a potential deadlock situation is detected, this method will fail and return a value of zero.

Every time the **Lock** method is called, an internal lock counter is incremented, and the lock will not be released until the lock count drops to zero. This means that each call to the **Lock** method must be matched by an equal number of calls to the **Unlock** method. Failure to do so will result in the server becoming non-responsive as it remains in a locked state.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[IsLocked](#), [Unlock](#)

InternetServer::MatchHostName Method

```
BOOL MatchHostName(  
    LPCTSTR lpszHostName,  
    LPCTSTR lpszHostMask  
    BOOL bResolve  
);
```

The **MatchHostName** method matches a host name against one or more strings that may contain wildcards.

Parameters

lpszHostName

A pointer to a string which specifies the host name or IP address to match.

lpszHostMask

A pointer to a string which specifies one or more values to match against the host name. The asterisk character can be used to match any number of characters in the host name, and the question mark can be used to match any single character. Multiple values may be specified by separating them with a semicolon.

bResolve

A boolean value which specifies if the host name or IP address should be resolved when matching the host against the mask string. If this parameter is non-zero, two checks against the host mask string will be performed; once for the host name specified and once for its IP address. If this parameter is zero, then the match is made only against the host name string provided.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **MatchHostName** method provides a convenient way for an application to determine if a given host name matches one or more mask strings which may contain wildcard characters. For example, the host name could be "www.microsoft.com" and the host mask string could be "*.microsoft.com". In this example, the method would return a non-zero value indicating the host name matched the mask. However, if the mask string was "*.net" then the method would return zero, indicating that there was no match. Multiple mask values can be combined by separating them with a semicolon; for example, the mask "*.com;*.org" would match any host name in either the .com or .org top-level domains.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetAddress](#), [GetHostAddress](#), [GetHostName](#)

InternetServer::Peek Method

```
INT Peek(  
    SOCKET hSocket,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Peek(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

Read data from the client without removing it from the socket buffer.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpBuffer

Pointer to the buffer in which the data will be stored. This argument may be NULL, in which case no data is copied from the socket buffers, however the function will return the number of bytes available to read.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. If the *lpBuffer* parameter is not NULL, this value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes available to read from the socket. A return value of zero indicates that there is no data available to read at that time. If the function fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **Peek** method returns data that is available to read from the socket, up to the number of bytes specified. The data returned by this method is not removed from the socket buffers. It must be consumed by a subsequent call to the **Read** method. The return value indicates the number of bytes that can be read in a single operation, up to the specified buffer size. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

If no data is available to be read, the method will return a value of zero. To determine if there is data available to be read, use the **IsReadable** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cswsock10.h

Import Library: cswskv10.lib

See Also

[IsReadable](#), [IsWritable](#), [Read](#), [ReadLine](#), [Write](#), [WriteLine](#)

CIInternetServer::PreProcessEvent Method

```
virtual LONG PreProcessEvent(  
    SOCKET hServer,  
    UINT nClientId,  
    UINT nEventId,  
    DWORD dwError,  
    BOOL& bHandled  
);
```

A virtual method that is invoked for each event generated by the server.

Parameters

hServer

The server handle. The application should treat this as an opaque value that is only valid as long as the server is active. This value should not be stored by the application and the handle value will change if the server is restarted.

nClientId

An unsigned integer which uniquely identifies the client that has issued a request to the server. This value is guaranteed to be unique to the client session throughout the life of the server and is never reused. The application should never make assumptions about the order in which IDs are allocated to the client sessions.

nEventId

An unsigned integer which specifies which event occurred. For a list of events, see [Server Event Constants](#).

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

bHandled

An integer which specifies if the event has been handled by the application. If this parameter is set to a non-zero value, the default event handler will not be invoked for the event.

Return Value

The method should return a value of zero to indicate that the default event handler should be invoked for the event. If the method returns a non-zero value, this value is passed back to the event dispatcher and the default handler will not be invoked.

Remarks

The **PreProcessEvent** method is invoked for each event that is generated, prior to the default handler for that event. To implement an event handler, the application should create a class derived from the **CIInternetServer** class, and then override this method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cswsock10.h

Import Library: cswskv10.lib

CInternetServer::Read Method

```
INT Read(  
    SOCKET hSocket,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Read(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

Read data from the client.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpBuffer

Pointer to the buffer in which the data will be stored. This parameter cannot be NULL.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes read. A return value of zero indicates that the client has closed the connection and there is no more data available to be read. If the method fails, the return value is `INET_ERROR`. To get extended error information, call the **GetLastError** method.

Remarks

The **Read** method will read up to the specified number of bytes and store the data in the buffer provided by the caller. If there is no data available to be read at the time this method is invoked, the session thread will block until at least one byte of data becomes available, the timeout period elapses or an error occurs. This method will return if any amount of data is sent by the client, and will not block until the entire buffer has been filled.

The application should never make an assumption about the amount of data that will be available to read. TCP considers all data to be an arbitrary stream of bytes and does not impose any structure on the data itself. For example, if the client is sending data to the server in fixed 512 byte blocks of data, it is possible that a single call to the **Read** method will return only a partial block of data, or it may return multiple blocks combined together. It is the responsibility of the application to buffer and process this data appropriately.

For applications that are built using the Unicode character set, it is important to note that the buffer is an array of bytes, not characters. If the client is sending string data to the server, it must be read as a stream of bytes and converted using the **MultiByteToWideChar** function. If the client is sending lines of text terminated with a linefeed or carriage return and linefeed pair, the **ReadLine** method will return a line of text at a time and perform this conversion for you.

To determine if there is data available to be read without causing the session thread to block, call the **IsReadable** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[IsReadable](#), [IsWritable](#), [Peek](#), [ReadLine](#), [Write](#), [WriteLine](#)

CInternetServer::ReadLine Method

```
BOOL ReadLine(  
    SOCKET hSocket,  
    LPTSTR lpszBuffer,  
    LPINT lpnLength  
);  
  
BOOL ReadLine(  
    LPTSTR lpszBuffer,  
    LPINT lpnLength  
);  
  
BOOL ReadLine(  
    SOCKET hSocket,  
    CString& strBuffer,  
    INT nMaxLength  
);  
  
BOOL ReadLine(  
    CString& strBuffer,  
    INT nMaxLength  
);
```

Read up to a line of data from the socket and return it in a string buffer.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpszBuffer

Pointer to the string buffer that will contain the data when the method returns. The string will be terminated with a null character, and will not contain the end-of-line characters. An alternate form of the method accepts a **CString** argument which will contain the line of text returned by the server.

lpnLength

A pointer to an integer value which specifies the length of the buffer. The value should be initialized to the maximum number of characters that can be copied into the string buffer, including the terminating null character. When the method returns, its value will updated with the actual length of the string.

nMaxLength

An integer value which specifies the maximum length of the buffer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ReadLine** method reads data from the socket and copies into a specified string buffer. Unlike the **Read** method which reads arbitrary bytes of data, this method is specifically designed to return a single line of text data in a string. When an end-of-line character sequence is

encountered, the method will stop and return the data up to that point. The string buffer is guaranteed to be null-terminated and will not contain the end-of-line characters. This method will force the thread to block until an end-of-line character sequence is processed, the read operation times out or the remote host closes its end of the socket connection.

There are some limitations when using **ReadLine**. The method should only be used to read text, never binary data. In particular, the method will discard nulls, linefeed and carriage return control characters. The Unicode version of this method will return a Unicode string, however it does not support reading raw Unicode data from the socket. Any data read from the socket is internally buffered as octets (eight-bit bytes) and converted to Unicode using the **MultiByteToWideChar** function.

The **Read** and **ReadLine** method calls can be intermixed, however be aware that **Read** will consume any data that has already been buffered by the **ReadLine** method and this may have unexpected results.

Unlike the **Read** method, it is possible for data to be returned in the string buffer even if the return value is zero. Applications should check the length of the string to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the return value.

Example

```
CString strBuffer;
BOOL bResult;

do
{
    bResult = pServer->ReadLine(strBuffer);

    if (strBuffer.GetLength() > 0)
    {
        // Process the line of data returned in the string
        // buffer; the string is always null-terminated
    }
} while (bResult);

DWORD dwError = pServer->GetLastError();

if (dwError == ST_ERROR_CONNECTION_CLOSED)
{
    // The remote host has closed its side of the connection and
    // there is no more data available to be read
}
else if (dwError != 0)
{
    // An error has occurred while reading a line of data
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IsReadable](#), [Peek](#), [Read](#), [ReadStream](#), [Write](#), [WriteLine](#), [WriteStream](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CIInternetServer::ReadStream Method

```
BOOL ReadStream(  
    SOCKET hSocket,  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwOptions,  
    LPBYTE lpMarker,  
    DWORD cbMarker  
);
```

```
BOOL ReadStream(  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwOptions,  
    LPBYTE lpMarker,  
    DWORD cbMarker  
);
```

Read a stream of data from the client and store it in the specified buffer.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpvBuffer

Pointer to the buffer that will contain or reference the data when the method returns. The actual argument depends on the value of the *dwOptions* parameter which specifies how the data stream will be stored.

lpdwLength

A pointer to an unsigned integer value which specifies the maximum length of the buffer and contains the number of bytes read when the method returns. This argument should always point to an initialized value. If the *lpvBuffer* argument specifies a memory buffer, then this argument cannot point to an initialized value of zero; if any other type of stream buffer is used and the initialized value is zero, that indicates that all available data from the socket should be returned until the end-of-stream marker is encountered or the remote host disconnects.

dwOptions

An unsigned integer value which specifies both the stream buffer type and any options to be used when reading the data stream. One of the following stream types may be specified:

Constant	Description
INET_STREAM_DEFAULT	The default stream buffer type is determined by the value passed as the <i>lpvBuffer</i> parameter. If the argument specifies a pointer to a global memory handle initialized to NULL, then the method will return a handle which references the data; otherwise, the method will consider the parameter a pointer to a block of pre-allocated memory which will contain the stream data when the

	method returns. In most cases, it is recommended that an application explicitly specify the stream buffer type rather than using the default value.
INET_STREAM_MEMORY	The <i>lpvBuffer</i> argument specifies a pointer to a pre-allocated block of memory which will contain the data read from the socket when the method returns. If this stream buffer type is used, the <i>lpdwLength</i> argument must point to an unsigned integer which has been initialized with the maximum length of the buffer.
INET_STREAM_HGLOBAL	The <i>lpvBuffer</i> argument specifies a pointer to a global memory handle. When the method returns, the handle will reference a block of memory that contains the stream data. The application should take care to make sure that the handle passed to the method does not currently reference a valid block of memory; it is recommended that the handle be initialized to NULL prior to calling this method.
INET_STREAM_HANDLE	The <i>lpvBuffer</i> argument specifies a Windows handle to an open file, console or pipe. This should be the same handle value returned by the CreateFile function in the Windows API. The data read from the socket will be written to this handle using the WriteFile function.
INET_STREAM_SOCKET	The <i>lpvBuffer</i> argument specifies a socket handle. The data read from the socket specified by the <i>hSocket</i> argument will be written to this socket. The socket handle passed to this method must have been created by this library; if it is a socket created by an third-party library or directly by the Windows Sockets API, you should either create another instance of the class and attach the socket using the AttachHandle method or use the INET_STREAM_HANDLE stream buffer type instead.

In addition to the stream buffer types listed above, the *dwOptions* parameter may also have one or more of the following bit flags set. Programs should use a bitwise operator to combine values.

Constant	Description
INET_STREAM_CONVERT	The data stream is considered to be textual and will be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the

	amount of data returned in the buffer to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.
INET_STREAM_UNICODE	The data stream should be converted to Unicode. This option should only be used with text data, and will result in the stream data being returned as 16-bit wide characters rather than 8-bit bytes. The amount of data returned will be twice the amount read from the source data stream; if the application is using a pre-allocated memory buffer, this must be considered before calling this method.

lpMarker

A pointer to an array of bytes which marks the end of the data stream. When this byte sequence is encountered by the method, it will stop reading and return to the caller. The buffer will contain all of the data read from the socket up to and including the end-of-stream marker. If this argument is NULL, then the method will continue to read from the socket until the maximum buffer size is reached, the remote host closes its socket or an error is encountered.

cbMarker

An unsigned integer value which specifies the length of the end-of-stream marker in bytes. If the *lpMarker* parameter is NULL, then this value must be zero.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ReadStream** method enables an application to read an arbitrarily large stream of data and store it in memory, write it to a file or even another socket. Unlike the **Read** method, which will return immediately when any amount of data has been read, **ReadStream** will only return when the buffer is full as specified by the *lpdwLength* parameter, the logical end-of-stream marker has been read, the socket closed by the remote host or when an error occurs. This method will force the session thread to block until the operation completes.

It is possible for data to be returned in the buffer even if the method returns a value of zero. Applications should also check the value of the *lpdwLength* argument to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the method return value.

Example

```
HGLOBAL hgb1Buffer = NULL; // Return data in a global memory buffer
DWORD cbBuffer = 102400; // Read up to 100K bytes
BOOL bResult;
```

```
bResult = pServer->ReadStream(&hgblBuffer, &cbBuffer,  
                              INET_STREAM_HGLOBAL | INET_STREAM_CONVERT);  
  
if (bResult && cbBuffer > 0)  
{  
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);  
  
    // Use data in the stream buffer  
  
    GlobalUnlock(hgblBuffer);  
    GlobalFree(hgblBuffer);  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[GetStreamInfo](#), [Read](#), [ReadLine](#), [StoreStream](#), [Write](#), [WriteLine](#), [WriteStream](#)

CIInternetServer::Reject Method

BOOL Reject();

Reject a pending client connection.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Reject** method rejects a pending client connection and the remote host will see this as the connection being aborted. If there are no pending client connections at the time, this method will immediately return with an error indicating that the operation would cause the server thread to block. This method should only be invoked from within an **OnAccept** event handler if the application wishes to reject the incoming connection before a client session is created.

To determine the IP address of a client that is attempting to connect to the server from within the **OnAccept** event, use the **GetClientAddress** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[GetClientAddress](#), [OnAccept](#)

CIInternetServer::Restart Method

```
BOOL Restart();
```

Restart the server, terminating all active client sessions.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Restart** method will restart the specified server, terminating all active client sessions. If the method is unable to restart the server for any reason, the server thread is terminated. The server retains all of the options specified for the previous instance.

If an application calls this method from within an event handler, the active client session (the client for which the event handler was invoked) may not get a disconnect notification. It is recommended that this method only be called by the same thread that created the server using the **Start** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cssock10.h

Import Library: csoskv10.lib

See Also

[Start](#), [Stop](#)

CInternetServer::Resume Method

BOOL Resume();

Resume accepting client connections.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Resume** method instructs the server to resume accepting new client connections after the **Suspend** method has been called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cswsock10.h

Import Library: csWSKV10.lib

See Also

[Restart](#), [Start](#), [Stop](#), [Suspend](#), [Throttle](#)

CIInternetServer::SetBacklog Method

```
BOOL SetBacklog(  
    UINT nBackLog  
);
```

Set the size of the backlog connection queue for the server.

Parameters

nBacklog

An integer value that specifies the size of the connection backlog queue.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero.

Remarks

The **SetBacklog** method specifies the size of the queue allocated for pending client connections. A value of zero specifies that the queue should be set to a reasonable default value. On Windows server platforms, the maximum value is large enough to queue several hundred pending connections. This method should only be called prior to invoking the **Start** method, it does not have any effect on an active server. To get the current size of the backlog queue, use the **GetBacklog** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetBacklog](#), [Start](#), [Data Members](#)

InternetServer::SetCertificate Method

```
BOOL SetCertificate(  
    DWORD dwProtocol,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName,  
    LPCTSTR lpszPassword  
);
```

```
BOOL SetCertificate(  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName,  
    LPCTSTR lpszPassword  
);
```

```
BOOL SetCertificate(  
    LPCTSTR lpszCertName,  
    LPCTSTR lpszPassword  
);
```

Specify the server certificate that should be used with secure connections.

Parameters

dwProtocol

An optional bitmask of supported security protocols. If this parameter is not specified, then a default set of security protocols will be automatically selected. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default selection of security protocols will be used when establishing a connection. The TLS 1.2, TLS 1.1 and TLS 1.0 protocols will be negotiated with the client, in that order of preference. This option will always request the latest version of the preferred security protocols and is the recommended value.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the client. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Note that SSL 2.0 has been deprecated and will never be used by default.
SECURITY_PROTOCOL_TLS	The TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the client. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some clients that reject any

attempt to use the older SSL protocol and require that only TLS be used.
--

lpzCertStore

An optional string value that specifies the name of the certificate store that contains the server certificate. If the certificate is stored in the registry, this parameter must specify a valid local certificate store name. If the certificate is stored in a file, this parameter should specify the full path to the file that contains the certificate. If this parameter is omitted, the personal certificate store for the current process will be used.

lpzCertName

A string value that specifies the name of the certificate. This parameter is required and cannot be NULL. Either the common name or the name assigned to the certificate may be specified. In most cases, this will be the fully qualified domain name of the host that the server is running on.

lpzPassword

An optional string value that specifies a password associated with the server certificate. This parameter is usually only required when the *lpzCertStore* parameter specifies a certificate stored in a file. If the server certificate does not have a password associated with it, this parameter is omitted.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `csksk10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableSecurity](#), [SetOptions](#), [Start](#)

InternetServer::SetClientData Method

```
BOOL SetClientData(  
    SOCKET hClient,  
    VOID LpvData  
);
```

```
BOOL SetClientData(  
    VOID LpvData  
);
```

Associate application defined data with the specified client session.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lppvData

Pointer to the application defined data associated with the specified client session.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates that application defined data for the client session could not be modified. To get extended error information, call the **GetLastError** method.

Remarks

The **SetClientData** method is used to associate application defined data with a specific client session. This is typically used to associate a pointer to a data structure or a class instance with the client socket. A pointer to the data can be retrieved using the **GetClientData** method.

You should never specify a pointer to a local variable or data structure that will go out of scope when the calling method exits. If you do this, the pointer will no longer be valid after the method exits and attempting to dereference that pointer at some later time can cause an exception to be thrown and terminate the program. You should always allocate a block of memory for the data using a method such as **HeapAlloc** or **LocalAlloc**. If you specify the address of a static or global data structure, you must use thread synchronization methods when dereferencing and modifying that structure.

Example

```
UINT *pnValue1 = new UINT;  
UINT *pnValue2 = NULL;  
  
*pnValue1 = 1234;  
  
if (!pServer->SetClientData(hClient, pnValue1))  
{  
    // Unable to associate the data with this session  
    return;  
}  
  
if (!pServer->GetClientData(hClient, (LPVOID *)&pnValue2))  
{
```

```
    // Unable to retrieve the data associated with this session
    return;
}

// pnValue2 == pnValue1
printf("The value of the user defined data is %u\n", *pnValue2);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[GetClientData](#)

CIInternetServer::SetClientMoniker Method

```
INT SetClientMoniker(  
    SOCKET hSocket,  
    LPCTSTR LpszMoniker  
);  
  
INT SetClientMoniker(  
    LPCTSTR LpszMoniker  
);
```

Associate a unique string alias with the specified client session.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpszMoniker

Pointer to a string which specifies the moniker for the specified client socket. If this parameter is NULL or specifies an empty string, a moniker will no longer be associated with the client session.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

A client moniker is a string which can be used to uniquely identify a specific client session aside from its socket handle. The **GetClientMoniker** method will return the moniker that was previously assigned to the client, if any. To obtain the socket handle associated with a given moniker, use the **FindClient** method.

Monikers are not case-sensitive, and they must be unique so that no client socket for a particular server can have the same moniker. The maximum length for a moniker is 127 characters.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cswsock10.h

Import Library: cswskv10.lib

See Also

[FindClient](#), [GetClientHandle](#), [GetClientId](#), [GetClientMoniker](#)

CInternetServer::SetLastError Method

```
VOID SetLastError(  
    DWORD dwErrorCode  
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SOCKET or INET_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cswsock10.h

Import Library: cswskv10.lib

See Also

[GetErrorString](#), [GetLastError](#), [ShowError](#)

CIInternetServer::SetOptions Method

```
BOOL SetOptions(  
    DWORD dwOptions  
);
```

Set one or more server options.

Parameters

dwOptions

An unsigned integer that specifies one or more option bitflags.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero.

Remarks

The **SetOptions** method sets the server options for the class instance. For a list of the available options, see [Server Option Constants](#). This method should only be called prior to invoking the **Start** method, it does not have any effect on an active server. The **GetOptions** method returns the options that are currently specified for the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[GetOptions](#), [Data Members](#)

InternetServer::SetPriority Method

```
INT SetPriority(  
    INT nPriority  
);
```

Set the priority assigned to the specified server.

Parameters

nPriority

An integer that specifies the server priority. It may be one of the following values:

Constant	Description
INET_PRIORITY_BACKGROUND (0)	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
INET_PRIORITY_LOW (1)	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
INET_PRIORITY_NORMAL (2)	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
INET_PRIORITY_HIGH (3)	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
INET_PRIORITY_CRITICAL (4)	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Return Value

If the method succeeds, the return value is the previous priority assigned to the server. If the method fails, the return value is INET_ERROR.

Remarks

The **SetPriority** method changes the current priority assigned to the specified server. Client connections that are accepted after this method is called will inherit the new priority as their default priority. Previously existing client connections will not be affected by this function.

Higher priority values increase the thread priority and processor utilization for each client session. You should only change the server priority if you understand the impact it will have on the system and have thoroughly tested your application. Configuring the server to run with a higher priority can have a negative effect on the performance of other programs running on the system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: csoskv10.lib

See Also

[GetPriority](#)

CInternetServer::SetStackSize Method

```
BOOL SetStackSize(  
    DWORD dwStackSize  
);
```

Change the initial size of the stack allocated for threads created by the server.

Parameters

dwStackSize

The amount of memory that will be committed to the stack for each thread created by the server. If this value is zero, a default stack size will be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetStackSize** method changes the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes. Increasing or decreasing the stack size will only affect new threads that are created by the server, it will not affect those threads that have already been created to manage active client sessions. It is recommended that most applications use the default stack size.

You should not change this value unless you understand the impact that it will have on your system and have thoroughly tested your application. Increasing the initial commit size of the stack will remove pages from the total system commit limit, and every page of memory that is reserved for stack cannot be used for any other purpose.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[GetStackSize](#), [Start](#)

CInternetServer::SetTimeout Method

```
INT SetTimeout(  
    UINT nTimeout  
);
```

Set the timeout interval used when waiting for a blocking operation to complete.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SetTimeout** method sets the amount of time that the server will wait for data to become available to read, and the default timeout for blocking network operations for each client session. If this method is invoked before the server has started, it will change the default timeout period for the entire server. If this method is invoked within a server event handler, it will change the timeout period for the active client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[GetTimeout](#), [IsReadable](#), [IsWritable](#), [OnTimeout](#)

CInternetServer::ShowError Method

```
INT ShowError(  
    LPCTSTR lpszAppTitle,  
    UINT uType,  
    DWORD dwErrorCode  
);
```

Display a message box which describes the specified error.

Parameters

lpszAppTitle

A pointer to a string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Remarks

The **ShowError** method will display a modal message box with an error message that corresponds to the specified error code. All top-level windows belonging to the current thread will be disabled until the user responds to the message box. An application should only invoke this method from within the main UI thread, never from within a server event handler such as **OnConnect**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetErrorString](#), [GetLastError](#), [SetLastError](#)

CInternetServer::Start Method

```
BOOL Start(  
    LPCTSTR lpszLocalHost,  
    UINT nLocalPort,  
    UINT nMaxClients,  
    DWORD dwOptions  
);
```

```
BOOL Start(  
    LPCTSTR lpszLocalHost,  
    UINT nLocalPort,  
    DWORD dwOptions  
);
```

```
BOOL Start(  
    UINT nLocalPort,  
    DWORD dwOptions  
);
```

```
BOOL Start(  
    UINT nLocalPort  
);
```

The **Start** method begins listening for client connections on the specified local address and port number. The server is started in its own thread and manages the client sessions independently of the calling thread. All interaction with the server and its client sessions takes place inside the class event handlers.

Parameters

lpszLocalHost

A pointer to a string which specifies the local hostname or IP address address that the server should be bound to. If this parameter is omitted or specifies a NULL pointer an appropriate address will automatically be used. If a specific address is used, the server will only accept client connections on the network interface that is bound to that address.

nLocalPort

The port number the server should use to listen for client connections. The port number used by the application must be unique and multiple instances of a server cannot use the same port number. It is recommended that a port number greater than 5000 be used for private, application-specific implementations.

nMaxClients

The maximum number of client connections that can be established with the server. A value of zero specifies that there should not be any fixed limit on the number of active client connections. This value can be adjusted after the server has been created by calling the **Throttle** method.

dwOptions

An unsigned integer value that specifies one or more options to be used when creating an instance of the server. For a list of the available options, see [Server Option Constants](#). If this parameter is omitted, the default options for the server instance will be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero.

To get extended error information, call **GetLastError**.

Remarks

In most cases, the *lpszLocalHost* parameter should be omitted or a NULL pointer. On a multihomed system, this will enable the server to accept connections on any appropriately configured network adapter. Specifying a hostname or IP address will limit client connections to that particular address. Note that the hostname or address must be one that is assigned to the local system, otherwise an error will occur.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the method will fail.

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

The server instance is managed in another thread, and all interaction with the server and active client connections are performed inside the event handlers. To disconnect all active connections, close the listening socket and terminate the server thread, call the **Stop** method.

Example

```
// EchoServer implementation
class CEchoServer : public CInternetServer
{
    void OnRead(SOCKET hSocket)
    {
        // Read data sent by the client to the server
        BYTE ioBuffer[1024];
        INT nBytesRead = Read(hSocket, ioBuffer, sizeof(ioBuffer));

        // Send a copy of the data back to the client
        if (nBytesRead > 0)
            Write(hSocket, ioBuffer, nBytesRead);
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    CEchoServer myServer;

    // Start the server listening for connections on port 7000
    if (myServer.Start(7000))
    {
        TCHAR szCommand[128], *pszCommand;

        // Process commands entered by the user at the console
        while (TRUE)
        {
            if ((pszCommand = _fgetts(szCommand, 128, stdin)) == NULL)
                break;

            if (_tcsicmp(pszCommand, _T("quit")) == 0)
                break;
        }
    }
}
```

```
        // Stop the server and terminate all clients
        myServer.Stop();
    }

    return 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cswsock10.h

Import Library: cswskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnumClients](#), [Restart](#), [Stop](#)

CInternetServer::Stop Method

BOOL Stop();

Stop the server, terminating all active client sessions and releasing the resources that were allocated for the server.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return a value of zero.

Remarks

The **Stop** method instructs the server to stop accepting client connections, disconnects all active client connections and terminates the thread that is managing the server session. The handle is no longer valid after the server has been stopped and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new server is started. An application should always consider the server handle to be opaque and never depend on it being a specific value.

If an application calls this method from within an event handler, the active client session (the client for which the event handler was invoked) may not get a disconnect notification. It is recommended that this method only be called by the same thread that created the server using the **Start** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cssock10.h

Import Library: cswskv10.lib

See Also

[Restart](#), [Start](#)

InternetServer::StoreStream Method

```
BOOL StoreStream(  
    LPCTSTR lpszFileName,  
    DWORD dwLength,  
    LPDWORD lpdwCopied  
    DWORD dwOffset,  
    DWORD dwOptions  
);
```

The **StoreStream** method reads the socket data stream and stores the contents in the specified file.

Parameters

lpszFileName

Pointer to a string which specifies the name of the file to create or overwrite.

dwLength

An unsigned integer which specifies the maximum number of bytes to read from the socket and write to the file. If this value is zero, then the method will continue to read data from the socket until the remote host disconnects or an error occurs.

lpdwCopied

A pointer to an unsigned integer value which will contain the number of bytes written to the file when the method returns.

dwOffset

An unsigned integer which specifies the byte offset into the file where the method will start storing data read from the socket. Note that all data after this offset will be truncated. A value of zero specifies that the file should be completely overwritten if it already exists.

dwOptions

An unsigned integer value which specifies one or more options. Programs can use a bitwise operator to combine any of the following values:

Constant	Description
INET_STREAM_CONVERT	The data stream is considered to be textual and will be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data written to the file to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.
INET_STREAM_UNICODE	The data stream should be converted to Unicode. This option should only be used with text data, and will result in the stream data being written as 16-bit

wide characters rather than 8-bit bytes. The amount of data returned will be twice the amount read from the source data stream. If the *dwOffset* parameter has a value of zero, the Unicode byte order mark (BOM) will be written to the beginning of the file.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **StoreStream** method enables an application to read an arbitrarily large stream of data and store it in a file. This method is essentially a simplified version of the **ReadStream** method, designed specifically to be used with files rather than memory buffers or handles.

Example

```
// Store all data sent by the client in a file
DWORD dwCopied = 0;
BOOL bResult = pServer->StoreStream(lpszFileName, 0, &dwCopied, 0,
INET_STREAM_CONVERT);

// Close the client connection to the server
pServer->Disconnect();

if (bResult && dwCopied > 0)
{
    // Process the data has been written to the file
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: csuskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Read](#), [ReadLine](#), [ReadStream](#), [Write](#), [WriteLine](#), [WriteStream](#)

CInternetServer::Suspend Method

```
BOOL Suspend();
```

Suspend the server and reject new client connections.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Suspend** method instructs the server to suspend accepting new client connections. Any incoming client connections will be rejected with an error message indicating that the server is currently unavailable. To resume accepting client connections, call the **Resume** method. Suspending the server will have no effect on clients that have already established a connection with the server.

It is recommended that you only suspend a server if absolutely necessary, and only for brief periods of time. If you want to limit the number of active client connections or control the connection rate for clients, use the **Throttle** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cswsock10.h

Import Library: cswskv10.lib

See Also

[Restart](#), [Resume](#), [Start](#), [Stop](#), [Throttle](#)

CIInternetServer::Throttle Method

```
BOOL Throttle(  
    UINT nMaxClients,  
    UINT nMaxClientsPerAddress,  
    DWORD dwConnectionRate  
);
```

The **Throttle** method limits the number of active client connections, connections per address and connection rate.

Parameters

nMaxClients

A value which specifies the maximum number of clients that may connect to the server. A value of zero specifies that there is no fixed limit to the number of client connections. A value of -1 specifies that the maximum number of clients should not be changed.

nMaxClientsPerAddress

A value which specifies the maximum number of clients that may connect to the server from the same IP address. A value of zero specifies that there is no fixed limit to the number of client connections per address. By default, there is a limit of four client connections per address. A value of -1 specifies that the maximum number of clients should not be changed.

dwConnectionRate

A value which specifies a restriction on the rate of client connections, limiting the number of connections that will be accepted within that period of time. A value of zero specifies that there is no restriction on the rate of client connections. The higher this value, the fewer the number of connections that will be accepted within a specific period of time. By default, there is no limit on the client connection rate. A value of -1 specifies that the connection rate should not be changed.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Throttle** method is used to limit the number of connections and the connection rate to minimize the potential impact of a large number of client connections over a short period of time. This can be used to protect the server from a client application that is malfunctioning or a deliberate denial-of-service attack in which the attacker attempts to flood the server with connection attempts.

If the maximum number of client connections or maximum number of connections per address is exceeded, the server will reject subsequent connection attempts until the number of active client sessions drops below the specified threshold. Note that adjusting these values lower than the current connection limits will not affect clients that have already connected to the server. For example, if the **Start** method is called with the maximum number of clients set to 100, and then **Throttle** is called lowering that value to 75, no existing client connections will be affected by the change. However, the server will not accept any new connections until the number of active clients drops below 75.

Increasing the connection rate value will force the server to slow down the rate at which it will accept incoming client connection requests. For example, setting this parameter to a value of 1000

would limit the server to accepting one client connection every second, while a value of 250 would allow the server to accept four client connections per second. Note that significantly increasing the amount of time the server must wait to accept client connections can exceed the connection backlog queue, resulting in client connections being rejected.

It is recommended that you always specify conservative connection limits for your server application based on expected usage. Allowing an unlimited number of client connections can potentially expose the system to denial-of-service attacks and should never be done for servers that are accessible over the Internet.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cswsock10.h

Import Library: cswskv10.lib

See Also

[Restart](#), [Resume](#), [Start](#), [Suspend](#)

CIInternetServer::Unlock Method

```
BOOL Unlock();
```

Unlock the server, allowing other client threads to resume execution.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **Unlock** method releases the lock on the specified server and allows any blocked threads to resume execution. Only one server may be locked at any one time, and only the thread which established the lock can unlock the server.

Every time the **Lock** method is called, an internal lock counter is incremented, and the lock will not be released until the lock count drops to zero. This means that each call to the **Lock** method must be matched by an equal number of calls to the **Unlock** method. Failure to do so will result in the server becoming non-responsive as it remains in a locked state.

The program should always check the return value from this method, and should never assume that the lock has been released. If a potential deadlock situation is detected, this method will fail and return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[IsLocked](#), [Lock](#)

CInternetServer::Write Method

```
INT Write(  
    SOCKET hSocket,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Write(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

Send the contents of the specified buffer to to the client.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the client.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is `INET_ERROR`. To get extended error information, call the **GetLastError** method.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[IsReadable](#), [IsWritable](#), [Read](#), [ReadLine](#), [WriteLine](#)

CInternetServer::WriteLine Method

```
BOOL WriteLine(  
    SOCKET hSocket,  
    LPCTSTR lpszBuffer,  
    LPINT lpnLength  
);
```

```
BOOL WriteLine(  
    LPCTSTR lpszBuffer,  
    LPINT lpnLength  
);
```

Write a line of data to the client, terminated with a carriage-return and linefeed.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpszBuffer

The pointer to a string buffer which contains the data that will be sent to the remote host. All characters up to, but not including, the terminating null character will be written to the socket. The data will always be terminated with a carriage-return and linefeed control character sequence. If this parameter points to an empty string or NULL pointer, then only a carriage-return and linefeed are written to the socket.

lpnLength

A pointer to an integer value which will contain the number of characters written to the socket, including the carriage-return and linefeed sequence. If this information is not required, a NULL pointer may be specified.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **WriteLine** method writes a line of text to the remote host and terminates the line with a carriage-return and linefeed control character sequence. Unlike the **Write** method which writes arbitrary bytes of data to the socket, the **WriteLine** method is specifically designed to write a single line of text data from a null-terminated string. This method will force the session thread to block until the complete line of text has been written, the write operation times out or the remote host aborts the connection.

There are some limitations when using **WriteLine**. This method should only be used to send text, never binary data. In particular, it will discard nulls and append linefeed and carriage return control characters to the data stream. The Unicode version of this method will accept a Unicode string, however it does not support writing raw Unicode data to the socket. Unicode strings will be automatically converted to UTF-8 encoding using the **WideCharToMultiByte** function and then written to the socket as a stream of bytes.

The **Write** and **WriteLine** methods can be safely intermixed.

Unlike the **Write** method, it is possible for data to have been written to the socket if the return value is zero. For example, if a timeout occurs while the method is waiting to send more data to the remote host, it will return zero; however, some data may have already been written prior to the error condition. If this is the case, the *lpnLength* argument will specify the number of characters actually written up to that point.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IsWritable](#), [Read](#), [ReadLine](#), [ReadStream](#), [Write](#), [WriteStream](#)

CInternetServer::WriteStream Method

```
BOOL WriteStream(  
    SOCKET hSocket,  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwOptions  
);
```

```
BOOL WriteStream(  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwOptions  
);
```

Write a stream of data to the client.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpvBuffer

Pointer to the buffer that contains or references the data to be written to the socket. The actual argument depends on the value of the *dwOptions* parameter which specifies how the data stream will be accessed.

lpdwLength

A pointer to an unsigned integer value which specifies the size of the buffer and contains the number of bytes written when the method returns. This argument should always point to an initialized value. If the *lpvBuffer* argument specifies a memory buffer or global memory handle, then this argument cannot point to an initialized value of zero.

dwOptions

An unsigned integer value which specifies the stream buffer type to be used when writing the data stream to the socket. One of the following stream types may be specified:

Constant	Description
INET_STREAM_DEFAULT	The default stream buffer type is determined by the value passed as the <i>lpvBuffer</i> parameter. If the argument specifies a global memory handle, then the method will write the data referenced by that handle; otherwise, the method will consider the parameter a pointer to a block of memory which contains data to be written. In most cases, it is recommended that an application explicitly specify the stream buffer type rather than using the default value.
INET_STREAM_MEMORY	The <i>lpvBuffer</i> argument specifies a pointer to a block of memory which contains the data to be written to the socket. If this stream buffer type is

	used, the <i>lpdwLength</i> argument must point to an unsigned integer which has been initialized with the size of the buffer.
INET_STREAM_HGLOBAL	The <i>lpvBuffer</i> argument specifies a global memory handle that references the data to be written to the socket. The handle must have been created by a call to the GlobalAlloc or GlobalReAlloc function. If this stream buffer type is used, the <i>lpdwLength</i> argument must point to an unsigned integer which has been initialized with the size of the buffer.
INET_STREAM_HANDLE	The <i>lpvBuffer</i> argument specifies a Windows handle to an open file, console or pipe. This should be the same handle value returned by the CreateFile function in the Windows API. The data read using the ReadFile function with this handle will be written to the socket.
INET_STREAM_SOCKET	The <i>lpvBuffer</i> argument specifies a socket handle. The data read from the socket specified by this handle will be written to the socket specified by the <i>hSocket</i> parameter. The socket handle passed to this method must have been created by this library; if it is a socket created by an third-party library or directly by the Windows Sockets API, you should either create another instance of the class and attach the socket using the AttachHandle method or use the INET_STREAM_HANDLE stream buffer type instead.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **WriteStream** method enables an application to write an arbitrarily large stream of data from memory or a file to the specified socket. Unlike the **Write** method, which may not write all of the data in a single method call, **WriteStream** will only return when all of the data has been written or an error occurs. This method will force the thread to block until the operation completes.

It is possible for some data to have been written even if the method returns a value of zero. Applications should also check the value of the *lpdwLength* argument to determine if any data was sent. For example, if a timeout occurs while the method is waiting to write more data, it will return zero; however, some data may have already been written to the socket prior to the error condition.

Example

```
CFile *pFile = new CFile();
DWORD dwLength = 0;

if (!pFile->Open(strFileName, CFile::modeRead | CFile::shareDenyWrite))
```

```
        return;

    dwLength = pFile->GetLength();

    if (dwLength > 0)
    {
        BOOL bResult = pServer->WriteStream(
            pFile->m_hFile,
            &dwLength,
            INET_STREAM_HANDLE);
    }

    delete pFile;
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[Read](#), [ReadLine](#), [ReadStream](#), [StoreStream](#), [Write](#), [WriteLine](#)

CInternetServer Event Handlers

Method	Description
OnAccept	The client is attempting to establish a connection to the server
OnConnect	The client has established a connection to the server
OnDisconnect	The client has disconnected from the server
OnError	The event handler encountered an error when processing a client event
OnRead	The client has sent data to the server
OnTimeout	The client has not sent data within the specified timeout period
OnWrite	The client is ready to receive data from the server

CInternetServer::OnAccept Method

```
virtual void OnAccept(  
    SOCKET hSocket  
);
```

A virtual method that is invoked when a client attempts to connect to the server.

Parameters

hSocket

A handle to the server socket.

Return Value

None.

Remarks

The **OnAccept** event handler is invoked when a client attempts to connect to the server, but prior to the connection being accepted. To implement an event handler, the application should create a class derived from the **CInternetServer** class, and then override this method.

This event only occurs before the server has checked the active client limits. This event is typically used to reject a connection based on some criteria established by the server, such as the IP address of the client attempting to make the connection. To obtain the IP address of the client that is attempting to connect to the server, use the **GetClientAddress** method using the server handle.

If this event handler is not implemented, the server will permit the client connection to complete. To reject the connection attempt, call the **Reject** method using the handle to the server socket. Rejecting the client connection within the **OnAccept** event handler may cause unexpected behavior by the client application because the connection process will not complete normally. Instead of rejecting the client connection within the **OnAccept** handler, it is recommended that most server applications implement an **OnConnect** event handler, perform any required checks and then gracefully disconnect the client using the **Disconnect** method if needed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cssock10.h

Import Library: csuskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Disconnect](#), [OnConnect](#), [OnDisconnect](#)

CInternetServer::OnConnect Method

```
virtual void OnConnect(  
    SOCKET hSocket,  
    UINT nClientId,  
    LPCTSTR lpszAddress,  
    UINT nPort  
);
```

A virtual method that is invoked after the client has connected to the server.

Parameters

hSocket

A handle to the client socket.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszAddress

A string that specifies the IP address of the client. This address may either be in IPv4 or IPv6 format, depending on how the server was configured and the address the client used to establish the connection.

nPort

An integer that specifies the port number that the client socket is bound to.

Return Value

None.

Remarks

The **OnConnect** event handler is invoked after the client has connected to the server. To implement an event handler, the application should create a class derived from the **CInternetServer** class, and then override this method.

This event only occurs after the server has checked the active client limits and the TLS handshake has been performed, if security has been enabled. If the server has been suspended, or the limit on the maximum number of client sessions has been exceeded, the server will reject the connection prior to this event handler being invoked.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cssock10.h

Import Library: cswskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Disconnect](#), [OnAccept](#), [OnDisconnect](#)

CInternetServer::OnDisconnect Method

```
virtual void OnDisconnect(  
    SOCKET hSocket  
);
```

A virtual method that is invoked when a client disconnects from the server.

Parameters

hSocket

A handle to the client socket.

Return Value

None.

Remarks

The **OnDisconnect** event handler is invoked when a client disconnects from the server, immediately before the client session is terminated. To implement an event handler, the application should create a class derived from the **CInternetServer** class, and then override this method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cswsock10.h

Import Library: cswskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Disconnect](#), [OnConnect](#)

CInternetServer::OnError Method

```
virtual void OnConnect(  
    SOCKET hSocket,  
    UINT nEventId,  
    DWORD dwError  
);
```

A virtual method that is invoked when the server encounters an error while handling a client request.

Parameters

hSocket

An unsigned integer which uniquely identifies the client session.

nEventId

An unsigned integer which identifies the client event that was being processed when the error occurred. For a list of event identifiers, see [Server Event Constants](#).

dwError

An unsigned integer value that specifies the error code.

Return Value

None.

Remarks

The **OnError** event handler is invoked whenever an error occurs while an event is being processed by the server. To implement an event handler, the application should create a class derived from the **CInternetServer** class, and then override this method.

It is important to note that this event is not raised for every error that occurs. The event only occurs when another event is being processed and an unhandled error occurs that must be reported back to the server application. The following are some common situations in which this event handler may be invoked:

- A network error occurs when the client connection is being accepted by the server. This could be the result of an aborted connection or some other lower-level failure reported by the networking subsystem on the server.
- The server is configured to use implicit SSL but cannot obtain the security credentials required to create the security context for the session. Usually this indicates that the server certificate cannot be found, or the certificate does not have a private key associated with it. It could also indicate a general problem with the cryptographic subsystem where the client and server could not successfully negotiate a cipher suite.
- Network errors that may occur when attempting to buffer data sent by the client. This usually indicates that the connection to the client has been aborted, either because the client is not acknowledging the data that has been exchanged with the server, or the client has terminated abnormally. This event will not occur if the client terminates the connection normally.

In most situations where this event handler is invoked, the error is not recoverable and the only action that can be taken is to terminate the client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cssock10.h

Import Library: cswskv10.lib

See Also

[OnConnect](#), [OnDisconnect](#), [OnTimeout](#)

CInternetServer::OnRead Method

```
virtual void OnRead(  
    SOCKET hSocket  
);
```

A virtual method that is invoked when a client sends data to the server.

Parameters

hSocket

A handle to the client socket.

Return Value

None.

Remarks

The **OnRead** event handler is invoked when a client sends data to the server. To implement an event handler, the application should create a class derived from the **CInternetServer** class, and then override this method. All server applications must implement an **OnRead** event handler to process the data sent by the client.

This event occurs whenever there is data available to be read from the client. The server application reads the data using the **Read** or **ReadLine** method, and can then send data back to the client using the **Write** or **WriteLine** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cswsock10.h

Import Library: cswskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Read](#), [ReadLine](#), [Write](#), [WriteLine](#), [OnWrite](#)

CInternetServer::OnTimeout Method

```
virtual void OnTimeout(  
    SOCKET hSocket  
);
```

A virtual method that is invoked when the client has not sent any data to the server within the timeout period.

Parameters

hSocket

A handle to the client socket.

Return Value

None.

Remarks

The **OnTimeout** event handler is invoked when a client has not sent any data to the server within the timeout period specified when the server was started. To implement an event handler, the application should create a class derived from the **CInternetServer** class, and then override this method.

This event handler is typically used to monitor the amount of time that a client is idle. The default timeout period for the server is 20 seconds, which would cause this event handler to be invoked whenever a client has not sent any data to the server in the last 20 seconds. The server may take no action, or it may disconnect the client after it has been idle for an extended period of time. To get the total amount of time that the client has been idle, call the **GetClientIdleTime** method. Note that while the server timeout period is specified in seconds, the **GetClientIdleTime** method returns the client idle time in milliseconds.

The default implementation for this event handler is to take no action. It is recommended that most server applications disconnect clients that are inactive. For typical client-server implementations that use transitory connections (where the client sends a single request to the server, the server responds and the connection is terminated) the amount of time that a client should be permitted to remain idle should be relatively low, usually 60 seconds or less. For persistent connections where there are multiple requests issued by the client over the lifetime of the session, a longer idle timeout period may be preferable.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cswsock10.h

Import Library: csWSKV10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetClientIdleTime](#)

CInternetServer::OnWrite Method

```
virtual void OnWrite(  
    SOCKET hSocket  
);
```

A virtual method that is invoked when the client is ready to receive data.

Parameters

hSocket

A handle to the client socket.

Return Value

None.

Remarks

The **OnWrite** event handler is invoked when a client is ready to receive data. To implement an event handler, the application should create a class derived from the **CInternetServer** class, and then override this method. All server applications must implement an **OnRead** event handler to process the data sent by the client.

This event occurs immediately after the **OnConnect** event and if security is enabled, after the TLS handshake has completed. It is used to notify the server application that the client is ready to receive data, and may be used to send an initial message to the client, typically identifying the server that it has connected to. In most cases, the **OnWrite** event handler will only be invoked once over the lifetime of the client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cswsock10.h

Import Library: cswskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Write](#), [WriteLine](#), [OnRead](#)

Internet Server Data Structures

- INETSTREAMINFO
- INTERNET_ADDRESS

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

INETSTREAMINFO Structure

This structure contains information about the data stream being currently read or written.

```
typedef struct _INETSTREAMINFO
{
    DWORD    dwStreamThread;
    DWORD    dwStreamSize;
    DWORD    dwStreamCopied;
    DWORD    dwStreamMode;
    DWORD    dwStreamError;
    DWORD    dwBytesPerSecond;
    DWORD    dwTimeElapsed;
    DWORD    dwTimeEstimated;
} INETSTREAMINFO, *LPINETSTREAMINFO;
```

Members

dwStreamThread

Specifies the numeric ID for the thread that created the socket.

dwStreamSize

The maximum number of bytes that will be read or written. This is the same value as the buffer length specified by the caller, and may be zero which indicates that no maximum size was specified. Note that if this value is zero, the application will be unable to calculate a completion percentage or estimate the amount of time for the operation to complete.

dwStreamCopied

The total number of bytes that have been copied to or from the stream buffer.

dwStreamMode

A numeric value which specifies the stream operation that is current being performed. It may be one of the following values:

Constant	Description
INET_STREAM_READ	Data is being read from the socket and stored in the specified stream buffer.
INET_STREAM_WRITE	Data is being written from the specified stream buffer to the socket.

dwStreamError

The last error that occurred when reading or writing the data stream. If no error has occurred, this value will be zero.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the operation is completed. This is based on the average number of bytes transferred per second and requires that a maximum stream buffer size be specified by the caller.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

See Also

[ReadStream](#), [StoreStream](#), [WriteStream](#)

INTERNET_ADDRESS Structure

This structure represents a numeric IPv4 or IPv6 address in network byte order.

```
typedef struct _INTERNET_ADDRESS
{
    INT    ipFamily;
    BYTE   ipNumber[16];
} INTERNET_ADDRESS, *LPINTERNET_ADDRESS;
```

Members

ipFamily

An integer which identifies the type of IP address. It will be one of the following values:

Constant	Description
INET_ADDRESS_UNKNOWN	The address has not been specified or the bytes in the <i>ipNumber</i> array does not represent a valid address. Functions which populate this structure will use this value to indicate that the address cannot be determined.
INET_ADDRESS_IPV4	Specifies that the address is in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address is in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

ipNumber

A byte array which contains the numeric form of the IP address. This array is large enough to store both IPv4 (32 bit) and IPv6 (128 bit) addresses. The values are stored in network byte order.

Remarks

The **INTERNET_ADDRESS** structure is used by some functions to represent an Internet address in a binary format that is compatible with both IPv4 and IPv6 addresses. Applications that use this structure should consider it to be opaque, and should not modify the contents of the structure directly.

For compatibility with legacy applications that expect an IP address to be 32 bits and stored in an unsigned integer, you can copy the first four bytes of the *ipNumber* array using the **CopyMemory** function or equivalent. Note that if this is done, your application should always check the *ipFamily* member first to make sure that it is actually an IPv4 address.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cswsock10.h

Mail Message Class Library

Compose and parse standard MIME formatted email messages.

Reference

- [Class Methods](#)
- [Error Codes](#)

Library Information

Class Name	CMailMessage
File Name	CSMSGV10.DLL
Version	10.0.1468.2518
LibID	E6D79220-7873-4170-BF39-5D7A0049655C
Import Library	CSMSGV10.LIB
Dependencies	None
Standards	RFC 822, RFC 2045, RFC 2046, RFC 2047, RFC 2048

Overview

The Mail Message class provides an interface for composing and processing email messages and newsgroup articles which are structured according to the Multipurpose Internet Mail Extensions (MIME) standard. Using this class, an application can easily create complex messages which include multiple alternative content types, such as plain text and styled HTML text, file attachments and customized headers.

It is not required that the developer understand the complex MIME standard; a single method call can be used to create multipart message, complete with a styled HTML text body and support for international character sets. The Mail Message class can be easily integrated with the other mail related protocol libraries, making it extremely easy to create and process MIME formatted messages.

SocketTools also includes a class for managing a local message storage file that can be used to store and retrieve multiple messages. Methods are provided to open and create storage files, add, remove and extract messages from storage, and search the stored messages for specific header field values. For more information, refer to the [CMessageStore](#) class.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-

bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Mail Message Class Methods

Class	Description
CMailMessage	Constructor which initializes the current instance of the class
~CMailMessage	Destructor which releases resources allocated by the class
Method	Description
AppendText	Append text to the body of the current message part
AttachData	Attach the contents of a buffer to the specified message
AttachFile	Attach a file to the specified message
AttachHandle	Attach the specified message handle to this instance of the class
ClearMessage	Clear the specified message, deleting all message parts
ClearText	Clear the body of the current message part
CompareText	Compare text in the body of the current message part
ComposeMessage	Compose a new message using the specified parameters
CopyMessage	Copy the contents of another message into the current message
CreatePart	Create a new message part for the specified message
DecodeText	Decode a base64 or quoted-printable encoded string
DeleteHeader	Delete the specified header field from the message
DeletePart	Delete the specified message part
DetachHandle	Detach the handle for the current instance of this class
EncodeText	Encode a string using base64 or quoted-printable encoding
EnumAttachments	Enumerate all file attachments in the current message
EnumHeaders	Enumerate all header fields in the current message part
EnumRecipients	Enumerate addresses of all message recipients
ExportMessage	Export the current message to a string buffer or text file
ExtractAllFiles	Extract all file attachments in the message and store them in the specified directory
ExtractFile	Extract the file attachment from the current message part
FindAttachment	Search for a file attachment in the specified message
FormatDate	Return a standard RFC 822 formatted date string
GetAllHeaders	Return the complete RFC 822 header values in a string buffer
GetAllRecipients	Return a comma-separated list of recipient addresses in a string buffer
GetAttachedFileName	Return the name of the file attachment for the current part
GetBoundary	Return the multipart message boundary string
GetContentDigest	Return encoded digest of message's content

GetContentLength	Return the length of the current message part content
GetDate	Return the date and time from the message header
GetErrorString	Return a description for the specified error code
GetExportOptions	Return a bitmask that describes current message export options
GetFileContentType	Return the content type for a specified file
GetFirstHeader	Return the first header field and value in the current message part
GetHandle	Return the message handle used by this instance of the class
GetHeader	Return the value of a specified header from the message
GetLastError	Return the last error code
GetMessageSize	Return the size of the complete message in bytes
GetNextHeader	Return the next header field and value in the current message part
GetPart	Return the current message part index
GetPartCount	Return the total number of message parts
GetSender	Return the email address of the message sender
GetText	Return the text of the current message part
ImportMessage	Import a message from the specified text file, clipboard or buffer
IsInitialized	Determine if the class has been successfully initialized
LocalizeText	Localize Unicode text to ANSI using a specific character set
ParseAddress	Parse the specified email address
ParseBuffer	Parse the specified text and add to the current message
ParseDate	Parse the specified RFC 822 formatted date string
ParseHeader	Parse the specified text and add to message header
SetExportOptions	Specify a bitmask that describes current message export options
SetFileContentType	Set the content type for a specific file name extension
SetLastError	Set the last error code
SetDate	Set the current date in the header for the specified message
SetHeader	Create or update a header field in the specified message
SetPart	Set the current message part index for the specified message
SetText	Create or update the specified message body
ShowError	Display a message box with a description of the specified error

CMailMessage::CMailMessage Method

`CMailMessage();`

The **CMailMessage** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **MimeInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[~CMailMessage](#), [IsInitialized](#)

CMailMessage::~CMailMessage

`~CMailMessage();`

The **CMailMessage** destructor releases resources allocated by the current instance of the **CMailMessage** object. It also uninitialized the library if there are no other concurrent uses of the class.

Remarks

When a **CMailMessage** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. All handles that were created for the session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CMailMessage](#)

CMailMessage::AppendText Method

```
LONG AppendText(  
    LPCTSTR lpszText  
);
```

The **AppendText** method appends the specified text to the body of the current message part.

Parameters

lpszText

A pointer to a string which specifies the text to be appended to the current message part.

Return Value

If the method succeeds, the return value is the number of bytes copied into the message. A return value of zero indicates that no text could be appended to the current message part. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ClearText](#), [CompareText](#), [GetText](#), [SetText](#)

CMailMessage::AttachData Method

```
BOOL AttachData(  
    LPBYTE lpBuffer,  
    LONG cbBuffer,  
    LPCTSTR lpszContentName,  
    LPCTSTR lpszContentType,  
    DWORD dwOptions  
);
```

```
BOOL AttachData(  
    CString& strBuffer,  
    LPCTSTR lpszContentName,  
    LPCTSTR lpszContentType,  
    DWORD dwOptions  
);
```

The **AttachData** method attaches the contents of the buffer to the message.

Parameters

lpBuffer

Pointer to a byte buffer which contains the data to be attached to the message. This parameter may be NULL, in which case no data is attached, but an additional empty message part will be created. An alternate form of this method accepts a **CString** object which contains the text to be attached to the message.

cbBuffer

An unsigned integer which specifies the number of bytes of data in the buffer pointed to by the *lpBuffer* parameter. If the *lpBuffer* parameter is NULL, this value must be zero.

lpszContentName

Pointer to a string which specifies a name for the data being attached to the message. This typically is used as a file name by the mail client to store the data in. If this parameter is NULL or an empty string then no name is defined and the data is attached as inline content. Note that if a file name is specified with a path, only the base name will be used.

lpszContentType

Pointer to a string which specifies the type of data being attached. The value must be a valid MIME content type. If this parameter is NULL or an empty string, then the buffer will be examined to determine what kind of data it contains. If there is only text characters, then the content type will be specified as "text/plain". If the buffer contains binary data, then the content type will be specified as "application/octet-stream", which is appropriate for any type of data.

dwOptions

A value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
MIME_ATTACH_DEFAULT	The data encoding is based on the content type. Text data is not encoded, and binary data is encoded using the standard base64 encoding algorithm.
MIME_ATTACH_BASE64	The data is always encoded using the standard base64 algorithm, even if the buffer only contains printable text

	characters.
MIME_ATTACH_UUCODE	The data is always encoded using the uuencode algorithm, even if the buffer only contains printable text characters.
MIME_ATTACH_QUOTED	The data is always encoded using the quoted-printable algorithm. This encoding should only be used if the data contains 8-bit text characters.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **AttachData** method allows an application to attach data to the message as either a file attachment or as inline content. The recipient of the message will see the attached data in the same way that they would see a file attached to the message using the **AttachFile** method.

If the specified message is not a multipart message, it is marked as multipart and the attached file is appended to the message. If the message is already a multipart message, an additional part is created and the attachment is added to the message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AttachFile](#), [ExportMessage](#), [ExtractFile](#), [GetAttachedFileName](#), [GetFileContentType](#), [ImportMessage](#), [SetFileContentType](#)

CMailMessage::AttachFile Method

```
BOOL AttachFile(  
    LPCTSTR lpszFileName,  
    DWORD dwOptions  
);
```

The **AttachFile** method attaches the specified file to the message.

Parameters

lpszFileName

Pointer to a string which specifies the name of the file to be attached to the message.

dwOptions

A value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
MIME_ATTACH_DEFAULT	The file attachment encoding is based on the file content type. Text files are not encoded, and binary files are encoded using the standard base64 encoding algorithm. This is the default option for file attachments.
MIME_ATTACH_BASE64	The file attachment is always encoded using the standard base64 algorithm, even if the attached file is a plain text file.
MIME_ATTACH_UUCODE	The file attachment is always encoded using the uuencode algorithm, even if the attached file is a plain text file.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

If the specified message is not a multipart message, it is marked as multipart and the attached file is appended to the message. If the message is already a multipart message, an additional part is created and the attachment is added to the message.

To attach data that is stored in a memory buffer rather than a file, use the **AttachData** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AttachData](#), [ExportMessage](#), [ExtractFile](#), [GetAttachedFileName](#), [GetFileContentType](#), [ImportMessage](#), [SetFileContentType](#)

CNetMessage::AttachHandle Method

```
VOID AttachHandle(  
    HMESSAGE hMessage  
);
```

The **AttachHandle** method attaches the specified message handle to the current instance of the class.

Parameters

hMessage

The handle to the message that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a message handle created outside of the class using the SocketTools API. Once the handle is attached to the class, the other class member functions may be used with that message.

If a message handle already has been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous message from being destroyed, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the *hMessage* parameter is presumed to be a valid message handle and no checks are performed to ensure that the handle is valid. Specifying an invalid message handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

See Also

[DetachHandle](#), [GetHandle](#)

CMailMessage::ClearMessage Method

```
BOOL ClearMessage();
```

The **ClearMessage** method clears the header and body of the specified message, and deletes all message parts.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

See Also

[ImportMessage](#), [ParseBuffer](#)

CMailMessage::ClearText Method

```
BOOL ClearText();
```

The **ClearText** method deletes the text from the body of the current message part.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AppendText](#), [CompareText](#), [GetText](#), [SetText](#)

CMailMessage::CompareText Method

```
BOOL CompareText(  
    LONG*lpnOffset,  
    LPCTSTR lpszBuffer,  
    LONG cchBuffer,  
    BOOL bCaseSensitive  
);
```

The **CompareText** method compares a text string against the contents of the current message part.

Parameters

lpnOffset

Pointer to a long integer which specifies the offset in the message at which to begin the comparison. This value will be updated when the method returns to indicate the offset position in the message where the comparison ended.

lpszBuffer

Pointer to a string buffer which contains the text that is to be compared against the body of the message.

cchBuffer

The number of characters in the buffer that should be compared against the body of the message.

bCaseSensitive

Boolean flag which specifies that the comparison should be case sensitive.

Return Value

If the text buffer matches the contents of the current message body, the method will return a non-zero value, and the *lpnOffset* argument will be set to position in the buffer where the match terminated. If the text buffer does not match, the method will return a value of zero, and the *lpnOffset* argument will be set to the position of the first non-matching character. The method will also return zero if one of the arguments is invalid. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AppendText](#), [ClearText](#), [GetText](#), [SetText](#)

CMailMessage::ComposeMessage Method

```
BOOL ComposeMessage(  
    LPCTSTR lpszFrom,  
    LPCTSTR lpszTo,  
    LPCTSTR lpszCc,  
    LPCTSTR lpszSubject,  
    LPCTSTR lpszMessageText,  
    LPCTSTR lpszMessageHTML,  
    UINT nCharacterSet,  
    UINT nEncodingType  
);
```

The **ComposeMessage** method creates a new message using the specified parameters.

Parameters

lpszFrom

A pointer to a string which specifies the sender's email address. This parameter may be NULL, in which case no sender address will be included in the message header.

lpszTo

A pointer to a string which specifies one or more recipient addresses. If multiple addresses are specified, each address must be separated by a comma. This parameter may be NULL, in which case no recipient addresses will be included in the message header.

lpszCc

A pointer to a string which specifies one or more addresses that will receive a copy of the message in addition to the listed recipients. If multiple addresses are specified, each address must be separated by a comma. This parameter may be NULL, in which case no carbon-copy addresses will be included in the message header.

lpszSubject

A pointer to a string which specifies the subject of the message. This parameter may be NULL, in which case no subject will be included in the message.

lpszMessageText

A pointer to a string which contains the body of the message as plain text. Each line of text contained in the string should be terminated with a carriage-return and linefeed (CRLF) pair, which is recognized as the end-of-line. If this parameter is NULL or points to an empty string, then the message will have an empty body unless the *lpszMessageHTML* parameter is not NULL.

lpszMessageHTML

A pointer to a string which contains the message using HTML formatting. If the *lpszMessageText* parameter is not NULL, then a multipart message will be created with both plain text and HTML text as the alternative. This allows mail clients to select which message body they wish to display. If the *lpszMessageText* argument is NULL or points to an empty string, then the message will only contain HTML. Although this is supported, it is not recommended because older mail clients may be unable to display the message correctly.

nCharacterSet

A numeric identifier which specifies the [character set](#) to use when composing the message. A value of zero specifies that the default UTF-8 character set should be used. It is recommended that you always use UTF-8 when composing a new message or creating a new message part.

nEncodingType

A numeric identifier which specifies the encoding type to use when composing the message. A value of zero specifies that default 7bit encoding should be used. The following values may also be used:

Constant	Description
MIME_ENCODING_7BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the most significant bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_ENCODING_8BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long and all bits are used. 8-bit encoding is typically used with multibyte character sets and is the default encoding used with Unicode text.
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.
MIME_ENCODING_BASE64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of data. This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

email addresses may be specified as simple addresses, or as commented addresses that include the sender's name or other information. For example, any one of these address formats are acceptable:

```
user@domain.tld
User Name <user@domain.tld>
user@domain.tld (User Name)
```

To specify multiple addresses, you should separate each address by a comma or semi-colon. Note that the *lpszFrom* parameter cannot specify multiple addresses, however it is permitted with the *lpszTo*, *lpszCc* and *lpszBcc* parameters.

To send a message that contains HTML, it is recommended that you provide both a plain text version of the message body and an HTML formatted version. While it is permitted to send a message that only contains HTML, some older mail clients may not be capable of displaying the message correctly. In some cases, anti-spam software will increase the spam score of messages that do not contain a plain text message body. This can result in your message being rejected or quarantined by the mail server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreatePart](#), [ExportMessage](#), [ImportMessage](#)

CMailMessage::CopyMessage Method

```
BOOL CopyMessage(  
    CMailMessage* pMessage  
);
```

The **CopyMessage** method copies the contents of the specified message.

Parameters

pMessage

A pointer to a CMailMessage object which contains the message to be copied. If this argument is NULL, the current message contents will be cleared.

Return Value

If the message was successfully copied, the method will return a non-zero value. If the method fails, it will return value of zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to create a copy of the specified message, replacing the current message contents. Note that this method creates a duplicate of the message referenced by the *pMessage* object, and any subsequent changes to the contents of the original message will not be reflected in the copy of that message.

Example

```
CMailMessage *pMessage1 = new CMailMessage();  
CMailMessage *pMessage2 = new CMailMessage();  
  
// Compose a test message  
pMessage1->ComposeMessage(_T("Bob Jones <bob@example.com>"),  
                          _T("Tom Smith <tom@example.com>"),  
                          NULL,  
                          _T("This is a test message"),  
                          _T("This is a test, this is only a test."));  
  
// Create a copy of the message and change the subject  
// The original subject in pMessage1 remains unchanged  
if (pMessage2->CopyMessage(pMessage1))  
{  
    CString &strSubject;  
    if (pMessage2->GetHeader(0, _T("Subject"), strSubject))  
    {  
        strSubject = _T("Re: ") + strSubject;  
        pMessage2->SetHeader(0, _T("Subject"), strSubject);  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

See Also

[ClearMessage](#), [ComposeMessage](#)

CMailMessage::CreatePart Method

```
INT CreatePart();  
  
INT CreatePart(  
    LPCTSTR lpszText,  
    UINT nCharacterSet,  
    UINT nEncodingType  
);
```

The **CreatePart** method creates a new part for the specified message. If this the first part created for a message that does not have the multipart content type specified, the message is marked as multipart and the header fields are updated.

Parameters

lpszText

A pointer to a string which specifies the text to be included in the body of the new message part. If this parameter is NULL or points to an empty string, no text is added to the message part.

nCharacterSet

A numeric identifier which specifies the [character set](#) to use when composing the message. A value of zero specifies the character set should be the same character set used to initially compose the message. It is recommended that you always use UTF-8 when composing a new message or creating a new message part.

nEncodingType

A numeric identifier which specifies the encoding type to use when composing the message. A value of zero specifies that default 7bit encoding should be used. The following values may also be used:

Constant	Description
MIME_ENCODING_7BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the most significant bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_ENCODING_8BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long and all bits are used. 8-bit encoding is typically used with multibyte character sets and is the default encoding used with Unicode text.
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.

MIME_ENCODING_BASE64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of data. This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text.
----------------------	--

Return Value

If the method succeeds, the return value is the new message part number. If the method fails, the return value is `MIME_ERROR`. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AttachFile](#), [ComposeMessage](#), [CreatePart](#), [DeletePart](#), [GetPart](#), [GetPartCount](#), [SetPart](#)

CMailMessage::DecodeText Method

```
LONG DecodeText(  
    LPCTSTR lpszInput,  
    LONG cchInput,  
    LPTSTR lpszOutput,  
    LONG cchOutput,  
    UINT nCharacterSet,  
    UINT nEncodingType  
);
```

```
LONG DecodeText(  
    LPCTSTR lpszInput,  
    LONG cchInput,  
    CString& strOutput,  
    UINT nCharacterSet,  
    UINT nEncodingType  
);
```

The **DecodeText** method decodes a string that was previously encoded using base64 or quoted-printable encoding.

Parameters

lpszInput

A pointer to a null terminated string which contains the encoded text. This parameter cannot be a NULL pointer.

cchInput

An integer value which specifies the number of characters of text in the input string which should be decoded. If this parameter is omitted or the value is -1, the entire length of the string up to the terminating null will be decoded.

lpszOutput

A pointer to a string buffer that will contain the decoded text. This buffer must be large enough to store all of the characters in the decoded text, including the terminating null character. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object instead of a fixed-length string buffer.

cchOutput

An integer value which specifies the maximum number of characters which can be copied into the output string buffer. The buffer must be large enough to store all of the decoded text and terminating null character. This value must be greater than zero. This parameter is not used if output buffer is specified as a **CString** object.

nCharacterSet

A numeric identifier which specifies the [character set](#) to use when decoding the input text. A value of zero specifies the character set is undefined and no Unicode text conversion is performed when the input string is decoded. If this value does not match the character set used when the text was originally encoded, the resulting output text may be invalid. This parameter may be omitted, in which case the method will default to using the UTF-8 character set.

nEncodingType

An integer value that specifies the encoding method used. This parameter may be omitted, or it may be one of the following values:

Constant	Description
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.
MIME_ENCODING_BASE64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of data. This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text. This is the default encoding type used by this method.

Return Value

If the input buffer can be successfully decoded, the return value is the length of the decoded output string. If the method returns zero, then no text was decoded and the output string buffer will be empty. If the method fails, the return value is `MIME_ERROR`. To get extended error information, call **GetLastError**.

Remarks

This method provides a means to decode text that was previously encoded using either base64 or quoted-printable encoding. In most cases, it is not necessary to use this method because the message parser will detect which character set and encoding was used, then automatically decode the message text if necessary.

The value of the *nCharacterSet* parameter does not affect the resulting output text, it is only used when decoding the input text. If the Unicode version of this method is called, the output text will be converted to UTF-16 and returned to the caller. If the ANSI version of this method is used, the decoded output will always be returned to the caller using the UTF-8 character set.

If the *nCharacterSet* parameter is specified as `MIME_CHARSET_UTF16`, the encoding type must be `MIME_ENCODING_BASE64`. Other encoding methods are not supported for Unicode strings and will cause the method to fail. In most cases, it is preferable to use `MIME_ENCODING_BASE64` as the encoding method, with quoted-printable encoding only used for legacy support.

If an unsupported encoding type is specified, this method will return `MIME_ERROR` and the output text string will be empty. This method cannot be used to decode uuencoded text.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EncodeText](#), [GetText](#), [SetText](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CMailMessage::DeleteHeader Method

```
BOOL DeleteHeader(  
    LPCTSTR lpszHeader  
);
```

The **DeleteHeader** method deletes the specified header field from the message.

Parameters

lpszHeader

Pointer to a string which specifies the header field that will be deleted.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero.

To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHeader](#), [SetHeader](#)

CMailMessage::DeletePart Method

```
BOOL DeletePart(  
    INT nMessagePart  
);
```

The **DeletePart** method deletes the specified message part from the multipart message. The memory allocated for the message part is released.

Parameters

nMessagePart

The message part index to delete.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method cannot be used to delete part zero, which is the main body of the message. Instead use the **Clear** method to clear the entire message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreatePart](#), [CreatePart](#), [GetPart](#), [GetPartCount](#), [ClearMessage](#), [SetPart](#)

CNetMessage::DetachHandle Method

```
HMESSAGE DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the message handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_MESSAGE` will be returned.

Remarks

This method is used to detach a message handle created by the class for use with the SocketTools API. Once the message handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

See Also

[AttachHandle](#), [GetHandle](#)

CMailMessage::EncodeText Method

```
LONG EncodeText(  
    LPCTSTR lpszInput,  
    LONG cchInput,  
    LPTSTR lpszOutput,  
    LONG cchOutput,  
    UINT nCharacterSet,  
    UINT nEncodingType  
);
```

```
LONG EncodeText(  
    LPCTSTR lpszInput,  
    LONG cchInput,  
    CString& strOutput,  
    UINT nCharacterSet,  
    UINT nEncodingType  
);
```

The **EncodeText** method encodes a string using base64 or quoted-printable encoding.

Parameters

lpszInput

A pointer to a null terminated string which contains the encoded text. This parameter cannot be a NULL pointer.

cchInput

An integer value which specifies the number of characters of text in the input string which should be encoded. If this parameter is omitted or the value is -1, the entire length of the string up to the terminating null will be encoded.

lpszOutput

A pointer to a string buffer that will contain the encoded text. This buffer must be large enough to store all of the characters in the encoded text, including the terminating null character. This parameter cannot be NULL. An alternate version of this method accepts a **CString** object instead of a fixed-length string buffer.

cchOutput

An integer value which specifies the maximum number of characters which can be copied into the output string buffer. The buffer must be large enough to store all of the encoded text and terminating null character. This value must be greater than zero.

nCharacterSet

A numeric identifier which specifies the [character set](#) to use when encoding the input text. A value of zero specifies the character set is undefined and no Unicode text conversion is performed when the input string is encoded.

nEncodingType

An integer value that specifies the encoding method used. It may be one of the following values:

Constant	Description
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented

	by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.
MIME_ENCODING_BASE64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of data. This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text.

Return Value

If the input buffer can be successfully encoded, the return value is the length of the encoded output string. If the method returns zero, then no text was encoded and the output string buffer will be empty. If the method fails, the return value is `MIME_ERROR`. To get extended error information, call **GetLastError**.

Remarks

This method provides a means to encode text using either base64 or quoted-printable encoding. It is not necessary to use this method to encode text when using the **SetText** method. The class will automatically encode message text which contains non-ASCII characters using the character set specified when the message is created.

If the *nCharacterSet* parameter is non-zero, the method will encode the text using the specified character set. If the Unicode version of this method is called, the input text is converted to ANSI using the code page associated with the character set. If the ANSI version of this method is called, the input text is converted to Unicode using the system default code page, and then back to ANSI using the specified character set.

If the *nCharacterSet* parameter specifies the `MIME_CHARSET_UTF16` character set, you must specify `MIME_ENCODING_BASE64` as the encoding method. Other encoding methods are not supported for Unicode strings and will cause the method to fail. It is not recommended you encode text as UTF-16 unless there is a specific requirement to use that character set.

It is recommended that you use the `MIME_CHARSET_UTF8` character set whenever possible. It is capable of encoding all Unicode code points, and is a standard for virtually all modern Internet applications. In most cases, it is preferable to use `MIME_ENCODING_BASE64` as the encoding method, with quoted-printable encoding only used for legacy support.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

CMailMessage::EnumAttachments Method

```
INT EnumAttachments(  
    LPCTSTR* lpFileNames,  
    INT nMaxFiles  
);
```

The **EnumAttachments** method enumerates all of the file attachments in the current message.

Parameters

lpFileNames

A pointer to an array of null-terminated strings that contain the names of the files attached to the message. If this parameter is NULL, the method will only return the number of files attached to the message.

nMaxFiles

An integer value that specifies the maximum size of the array of string pointers specified by the *lpFileNames* parameter. If this value is zero, the *lpFileNames* parameter is ignored and the method will only return the number of files attached to the message.

Return Value

If the method succeeds, the return value is the number of files attached to the message. If the message does not contain any file attachments, this method will return a value of zero. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Example

```
LPCTSTR lpszFiles[MAXFILES];  
  
INT nFiles = pMessage->EnumAttachments(lpszFiles, MAXFILES);  
if (nFiles == MIME_ERROR)  
{  
    DWORD dwError = pMessage->GetLastError();  
    _tprintf(_T("Unable to enumerate attachments, error 0x%08lx\n"), dwError);  
    return;  
}  
  
_tprintf(_T("There are %d files attached to the message\n"), nFiles);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AttachFile](#), [ExportMessage](#), [ExtractAllFiles](#), [GetAttachedFileName](#), [ImportMessage](#)

CMailMessage::EnumHeaders Method

```
INT EnumHeaders();

INT EnumHeaders(
    LPCTSTR* lpHeaderList,
    INT nMaxHeaders
);
```

The **EnumHeaders** method returns a list of pointers to all header field names in the current message part. This can be used in conjunction with the **GetHeader** method to retrieve the values for every header in the message.

Parameters

lpHeaderList

Pointer to an array of pointers to null terminated header field names. If this parameter is NULL, the method only returns the number of headers in the current message part.

nMaxHeaders

The maximum number of header fields which may be returned in the *lpHeaderList* parameter.

Return Value

If the method succeeds, the return value is the total number of headers that are defined in the current message part. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

The values returned in the header list array must not be directly modified by the application. There is no specific order in which the header fields are enumerated by this method. The header fields from an imported message may not be returned in the same order as which they appear in the message. An application should never make an assumption about the order in which one or more header fields are defined.

If this method is called without any arguments, it returns the number of headers in the current message part but does not return any data. This is useful for determining how much memory must be allocated for the *lpHeaderList* argument.

Example

```
// Determine the total number of headers in the current
// message part

nHeaders = pMessage->EnumHeaders();
if (nHeaders > 0)
{

    // Allocate memory for the list of headers

    lpHeaderList = (LPCTSTR *)malloc(nHeaders * sizeof(LPCTSTR));
    assert(lpHeaderList != NULL);

    // Retrieve the list of headers in the current
    // message part, and get their values

    pMessage->EnumHeaders(lpHeaderList, nHeaders);
    for (nIndex = 0; nIndex < nHeaders; nIndex++)
```

```
{  
  
    LPCTSTR lpszValue;  
    lpszValue = pMessage->GetHeader(lpHeaderList[nIndex]);  
    assert(lpszValue != NULL);  
  
    printf("%s: %s\n", lpHeaderList[nIndex], lpszValue);  
  
}  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFirstHeader](#), [GetHeader](#), [GetNextHeader](#), [SetHeader](#)

CMailMessage::EnumRecipients Method

```
INT EnumRecipients(  
    LPCTSTR lpszExtraAddress,  
    LPTSTR lpBuffer,  
    LPDWORD lpcchBuffer  
);  
  
INT EnumRecipients(  
    LPTSTR lpBuffer,  
    LPDWORD lpcchBuffer  
);  
  
INT EnumRecipients(  
    CStringArray& arrayRecipients,  
    LPCTSTR lpszExtraAddress  
);
```

The **EnumRecipients** method returns a null-terminated list of strings which contain the email address of each recipient for the specified message.

Parameters

lpszExtraAddress

A pointer to a string which contains one or more additional email addresses that should be included in the list, in addition to those found in the message. If more than one address is specified, each address should be separated by a comma. This parameter may be NULL if there are no extra addresses to include in the recipient list.

lpBuffer

Pointer to buffer which will contain zero or more null-terminated strings. The end of the string list is indicated by an additional terminating null. If this parameter is NULL, the method will calculate the minimum number of characters required to store the addresses and return the value in the *lpcchBuffer* parameter.

lpcchBuffer

A pointer to an unsigned integer which should be initialized to the maximum number of characters that can be copied into the buffer specified by the *lpBuffer* parameter. When the method returns, it will be updated to contain the actual number of characters copied into the buffer. If the *lpBuffer* parameter is NULL, then this value will contain the minimum number of characters required to store all of the recipient addresses in the current message.

arrayRecipients

A reference to a **CStringArray** object that will contain each of the recipient addresses specified in the message. This version of the method is only available if the program is compiled using Microsoft Foundation Classes (MFC).

Return Value

If the method succeeds, the return value is the total number of recipients for the current message. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnumRecipients** method returns a list of recipient email addresses for the specified message, with each address being terminated by a null character. The end of the list is indicated by an

additional null character. To determine the size of the buffer you should pass to this function, you can specify the *lpBuffer* parameter as NULL and initialize the value of the *lpcchBuffer* parameter to zero.

An alternative to the **EnumRecipients** method is the **GetAllRecipients** method that returns a comma-separated list of recipient addresses in a string buffer.

Example

```
LPTSTR lpRecipients = NULL;
DWORD cchRecipients = 0;
INT nRecipients = 0;

// Determine the number of characters that should be allocated to store
// all of the recipient addresses in the current message

nRecipients = pMessage->EnumRecipients(NULL, &cchRecipients);

// Allocate the memory for the string buffer that will contain all
// of the recipient addresses and call EnumRecipients
// again to store those addresses in the buffer

if (nRecipients > 0 && cchRecipients > 0)
{
    lpRecipients = (LPTSTR)::LocalAlloc(LPTR, (cchRecipients * sizeof(TCHAR)));
    if (lpRecipients == NULL)
        return; // Virtual memory exhausted

    nRecipients = pMessage->EnumRecipients(lpRecipients,
                                           &cchRecipients);
}

// Move through the buffer, processing each recipient address
// that was returned

if (nRecipients > 0)
{
    LPTSTR lpszAddress = lpRecipients;
    INT cchAddress;

    while (lpszAddress != NULL)
    {
        if ((cchAddress = lstrlen(lpszAddress)) == 0)
            break;

        // lpszAddress specifies a recipient address
        // Advance to the next address string in the buffer
        lpszAddress += cchAddress + 1;
    }
}

if (lpRecipients != NULL)
{
    LocalFree((HLOCAL)lpRecipients);
    lpRecipients = NULL;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetAllRecipients](#), [GetFirstHeader](#), [GetHeader](#), [GetNextHeader](#), [SetHeader](#)

CMailMessage::ExportMessage Method

```
BOOL ExportMessage(  
    LPCTSTR lpszFileName,  
    DWORD dwExportOptions  
);  
  
BOOL ExportMessage(  
    HGLOBAL* LphBuffer,  
    LPDWORD LpdwBufferSize,  
    DWORD dwExportOptions  
);  
  
BOOL ExportMessage(  
    LPTSTR lpszMessage,  
    DWORD dwMessageSize,  
    DWORD dwExportOptions  
);  
  
BOOL ExportMessage(  
    CString& strMessage,  
    DWORD dwExportOptions  
);
```

The **ExportMessage** method exports the message to a file, the system clipboard or global memory buffer.

Parameters

lpszFileName

A pointer to a string which specifies the file name that will contain the message. If the file already exists, it will be overwritten with the message contents.

lpszMessage

A pointer to a string which will contain the message. An alternate form of this method accepts a pointer to a global memory handle which will contain the message data when the method returns.

dwMessageSize

An unsigned integer value which specifies the maximum size of the *lpszMessage* buffer.

dwExportOptions

An unsigned integer which specifies how the message will be exported. The following values may be combined using a bitwise Or operator:

Constant	Description
MIME_OPTION_DEFAULT	The default export options. The headers for the message are written out in a specific consistent order, with custom headers written to the end of the header block regardless of the order in which they were set or imported from another message. If the message contains Bcc, Received, Return-Path, Status or X400-Received header fields, they will not be exported.
MIME_OPTION_KEEPPORDER	The original order in which the message header fields were set or imported are preserved when the message

	is exported.
MIME_OPTION_ALLHEADERS	All headers, including the Received, Return-Path, Status and X400-Received header fields will be exported. Normally these headers are not exported because they are only used by the mail transport system. This option can be useful when exporting a message to be stored on the local system, but should not be used when exporting a message to be delivered to another user.
MIME_OPTION_NOHEADERS	When exporting a message, the main header block will not be included. This can be useful when creating a multipart message for services which expect MIME formatted data, such as HTTP POST requests. This option should never be used for email messages being submitted using SMTP.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

If the version of this method is used which exports the message to a pre-allocated string buffer, the *dwMessageSize* parameter must specify the maximum size of the buffer. If the method succeeds, the message will be copied to the buffer. If the method fails, the previous contents of the buffer will not be preserved. If the buffer provided is not large enough to store the entire message, the message contents will be truncated. The **GetMessageSize** method can be used to determine the minimum size of the buffer required to store the complete message.

If the version of this method is used which returns an HGLOBAL memory handle to the caller, the handle must be dereferenced using the **GlobalLock** function. No changes should be made to this copy of the message. If you wish to modify the contents of the message, allocate a local buffer and make a copy of the message contents, or use the method which exports the message to a pre-allocated string buffer. Your application is responsible for calling **GlobalUnlock** and **GlobalFree** to unlock and free the handle when it is no longer needed.

Example

The following example exports the contents of a message to a global memory buffer:

```
HGLOBAL hgb1Message = NULL;
DWORD dwMessageSize = 0;

if (pMessage->Export(&hgb1Message, &dwMessageSize))
{
    LPBYTE lpMessage = (LPBYTE)GlobalLock(hgb1Message);

    if (lpMessage)
    {
        // Process the contents of the message
    }

    GlobalUnlock(hgb1Message);
    GlobalFree(hgb1Message);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImportMessage](#), [GetMessageSize](#), [SetExportOptions](#)

CMailMessage::ExtractAllFiles Method

```
INT ExtractAllFiles(  
    LPCTSTR lpszDirectory  
);
```

The **ExtractAllFiles** method extracts all of the file attachments in a message and stores them in the specified directory.

Parameters

lpszDirectory

A pointer to a string which specifies the name of the directory where the file attachments should be stored. If this parameter is NULL or points to an empty string, the attached files will be stored in the current working directory on the local system.

Return Value

If the method succeeds, the return value is the number of file attachments which were extracted from the message. If the message does not contain any file attachments, this method will return a value of zero. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method will extract all of the files that are attached to the message and store them in the specified directory. The directory must exist and the current user must have the appropriate permissions to create files there. If a file with the same name as the attachment already exists, it will be overwritten with the contents of the attachment. If the file attachment was encoded using base64 or uuencode, this function will automatically decode the contents of the attachment.

To determine the file names for each of the attachments in a message, use the **EnumAttachments** method. To store a file attachment on the local system using a name that is different than the file name of the attachment, use the **ExtractFile** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeAttachFile](#), [EnumAttachments](#), [ExportMessage](#), [ExtractFile](#), [GetAttachedFileName](#), [ImportMessage](#)

CMailMessage::ExtractFile Method

```
BOOL ExtractFile(  
    INT nMessagePart,  
    LPCTSTR lpszFileName  
);  
  
BOOL ExtractFile(  
    LPCTSTR lpszAttachment,  
    LPCTSTR lpszFileName  
);  
  
BOOL ExtractFile(  
    LPCTSTR lpszFileName  
);
```

The **ExtractFile** method extracts a file attachment from the message and stores it on the local system.

Parameters

nMessagePart

An integer value that specifies the message part that contains the file attachment. This value may be -1, in which case the current message part will be used.

lpszAttachment

A pointer to a string that specifies the file name for the attachment in the message. If the file name of the attachment is not known, this parameter can be NULL or point to an empty string.

lpszFileName

A pointer to a string that specifies the name of a file on the local system. If this parameter is NULL or points to an empty string, the value of the *lpszAttachment* parameter will specify the name of the file in the current working directory. If both the *lpszAttachment* and *lpszFileName* parameters are NULL, the method will fail.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method will store the contents of a file attachment in the current message part to the specified file on the local system. If a path is specified as part of the file name, it must exist and the current user must have the appropriate permissions to create the file. If a file with the same name already exists, it will be overwritten with the contents of the attachment. If the file attachment was encoded using base64 or uuencode, this method will automatically decode the contents of the attachment.

If the *nMessagePart* parameter is specified, then an attachment in that message part will be stored in the specified file. If the message part does not contain a file attachment, the method will fail.

If the *lpszAttachment* parameter is specified, the method will search the entire message for an attachment with the same file name. The search is not case-sensitive, however it must match the attachment file name completely. This method will not match partial file names or names that include wildcard characters. If a match is found, the contents of that attachment will be stored in the file specified by the *lpszFileName* parameter.

To extract all of the files attached to a message in a single method call, use the **ExtractAllFiles** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AttachFile](#), [EnumAttachments](#), [ExportMessage](#), [ExtractAllFiles](#), [ImportMessage](#)

CMailMessage::FindAttachment Method

```
INT FindAttachment(  
    LPCTSTR lpszFileName  
);
```

The **FindAttachment** method searches the message for an attachment with the specified file name.

Parameters

lpszFileName

Pointer to a string that specifies the name of the file attachment to search for. This parameter should only specify a base file name; it should not include a file path and cannot be NULL.

Return Value

If the method succeeds, the return value is the message part number that contains an attachment that matches the specified file name. If the message does not contain an attachment with the specified file name, the method will return MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method will search the message for a attachment that matches the specified file name. The search is not case-sensitive, however it must match the attachment file name completely. This method will not match partial file names or names that include wildcard characters. To obtain a list of all of the files attached to a message, use the **EnumAttachments** method.

Example

```
// The name of the file attachment to search for  
LPCTSTR lpszFileName = _T("MyProject.docx");  
  
// Search for the attached file and store it on the local system  
INT nMessagePart = pMessage->FindAttachment(lpszFileName);  
if (nMessagePart != MIME_ERROR)  
{  
    pMessage->SetPart(nMessagePart);  
  
    if (pMessage->ExtractFile(lpszFileName) != MIME_ERROR)  
        _tprintf(_T("Saved file attachment %s\n"), lpszFileName);  
    else  
    {  
        _tprintf(_T("Unable to save file attachment %s\n"), lpszFileName);  
        return;  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AttachFile](#), [ExtractAllFiles](#), [ExtractFile](#), [GetAttachedFileName](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CMailMessage::FormatDate Method

```
LPTSTR FormatDate(  
    LONG nSeconds,  
    LONG nTimezone,  
    LPTSTR lpszDate,  
    INT cchMaxDate  
);
```

```
LPTSTR FormatDate(  
    LONG nSeconds,  
    LPTSTR lpszDate,  
    INT cchMaxDate  
);
```

```
LPTSTR FormatDate(  
    LONG nSeconds,  
    LONG nTimezone,  
    CString& strDate  
);
```

```
LPTSTR FormatDate(  
    LONG nSeconds,  
    CString& strDate  
);
```

The **FormatDate** converts the specified date, expressed as the number of seconds since 1 January 1970, into a string compatible with the RFC 822 standard format for email messages.

Parameters

nSeconds

A long integer which specifies the number of seconds since 1 January 1970 00:00:00 UTC. This date is commonly called the epoch, and is the base date used by the standard C time methods. If the value of this parameter is zero, the current date and time is used.

nTimezone

A pointer to a long integer which is set to the difference in seconds between the specified date's timezone and Coordinated Universal Time. A value of zero specifies Coordinated Universal Time, while a positive value specifies a timezone west of UTC, and a negative value specifies a timezone east of UTC. For example, the Eastern timezone would be specified as the value 18000 and the Pacific timezone would be the value 28800. If this argument is omitted from the method, then the current timezone is used.

lpszDate

A buffer which will contain the formatted date as a null-terminated string. This parameter cannot be a NULL pointer.

cchMaxDate

The maximum number of characters, including the terminating null character, which may be copied into the date string buffer.

Return Value

If the method succeeds, a pointer to the date string buffer is returned. If the method fails, a NULL pointer is returned. To get extended error information, call **GetLastError**.

Remarks

The date string is returned in a standard format as outlined in RFC 822, the document which describes the basic structure of Internet email messages. This format is as follows:

www, dd mmm yyyy hh:mm:ss [-]zzzz

Each part of the date string is defined as follows:

Format	Description
www	Weekday
dd	Day
mmm	Month
yyyy	Year
hh	Hour (24-hour clock)
mm	Minutes
ss	Seconds
zzzz	Timezone

The weekday and month are displayed using standard three-character English abbreviations. The timezone is displayed as the difference (in hours and minutes) between the specified timezone and Coordinated Universal Time. For example, if the timezone is eight hours west of Coordinated Universal Time, the *nTimezone* value would be 28800. This would be displayed as -0800 in the formatted date string.

Note that the format of the date string is defined by the RFC 822 standard, and is not affected by localization settings on the host system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ParseDate](#), [GetDate](#), [SetDate](#)

CMailMessage::GetAllHeaders Method

```
INT GetAllHeaders(  
    LPTSTR lpszHeaders,  
    INT nMaxLength  
);  
  
INT GetAllHeaders(  
    CString& strHeaders  
);
```

The **GetAllHeaders** method returns the complete RFC 822 header values in a string buffer.

Parameters

lpszHeaders

Pointer to string buffer which will contain the header values for the current message. This parameter may be NULL, in which case the method will calculate the number of characters needed to store the complete header block.

nMaxLength

An integer value which specifies the maximum number of characters that can be stored in the *lpszHeaders* string. If the *lpszHeaders* parameter is NULL, this value must be zero. If the *lpszHeaders* parameter is not NULL, this value must be large enough to store the entire list of addresses.

strHeaders

A **CString** object that will contain the header values for the current message. This version of the method is available if MFC or ATL is being used by the application, and the memory required to store the complete header block will be calculated automatically.

Return Value

If the method succeeds and the *lpszHeaders* parameter is NULL, the return value is the minimum number of characters that should be allocated to store all of the header values, including the terminating null character. If the *lpszHeaders* parameter is not NULL, then the return value is the number of characters copied into the string, not including the terminating null character. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetAllHeaders** method will return all of the RFC 822 header values in a string buffer. This includes the message headers that are most commonly referred to, such as the To, From and Subject headers. Each header and its value are separated by a colon, and terminated with a carriage return and linefeed (CRLF) pair.

The headers and their values returned by this method will not be identical to the header block in the original message. If a header value is split across multiple lines, this method will fold the text, returning the complete header value on a single line of text and removing any extraneous whitespace. If the header value has been encoded by the mail client, this method will return the decoded value, not the original encoded value.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFirstHeader](#), [GetHeader](#), [GetNextHeader](#), [SetHeader](#)

CMailMessage::GetAllRecipients Method

```
INT GetAllRecipients(  
    LPCTSTR lpszExtraAddress,  
    LPTSTR lpszRecipients,  
    INT nMaxLength  
);  
  
INT GetAllRecipients(  
    CString& strRecipients,  
    LPCTSTR lpszExtraAddress  
);
```

The **GetAllRecipients** method returns a comma-separated list of recipient addresses in a string buffer.

Parameters

lpszExtraAddress

A pointer to a string which contains one or more additional email addresses that should be included in the list, in addition to those found in the message. If more than one address is specified, each address should be separated by a comma. This parameter may be NULL if there are no extra addresses to include in the recipient list.

lpszRecipients

Pointer to string buffer which will contain a comma-separated list of email addresses when the method returns. This parameter may be NULL, in which case the method will calculate the number of characters needed to store the complete list.

nMaxLength

An integer value which specifies the maximum number of characters that can be stored in the *lpszRecipients* string. If the *lpszRecipients* parameter is NULL, this value must be zero. If the *lpszRecipients* parameter is not NULL, this value must be large enough to store the entire list of addresses.

Return Value

If the method succeeds and the *lpszRecipients* parameter is NULL, the return value is the minimum number of characters that should be allocated to store the list recipient addresses, including the terminating null character. If the *lpszRecipients* parameter is not NULL, then the return value is the number of characters copied into the buffer, not including the terminating null character. If the method returns a value of zero, then the specified message has no recipients. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetAllRecipients** method is useful for creating a list of message recipients that can be passed to methods like **CSmtpClient::SendMessage**. If you wish to dynamically allocate the string buffer that will contain the list of recipients, then the *lpszRecipients* parameter should be NULL and the *nMaxLength* parameter should have a value of zero. The method will then return the recommended size of the buffer that should be allocated. This value is guaranteed to be large enough to store the entire list of message recipients, including the terminating null character.

Example

```
CString strRecipients;
```

```
if (pMessage->GetAllRecipients(strRecipients) < 1)
{
    // There are no recipients for the current message
    AfxMessageBox(IDS_ERROR_NO_RECIPIENTS);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnumRecipients](#), [GetFirstHeader](#), [GetHeader](#), [GetNextHeader](#), [SetHeader](#)

CMailMessage::GetAttachedFileName Method

```
BOOL GetAttachedFileName(  
    LPTSTR lpszFileName,  
    INT cchFileName  
);  
  
BOOL GetAttachedFileName(  
    CString& strFileName  
);
```

The **GetAttachedFileName** method returns the file name for the attachment to the current message part.

Parameters

lpszFileName

Pointer to a buffer that will contain the current attachment file name as a string. An alternate form of this method accepts a **CString** object which will contain the file name when the method returns.

cchFileName

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The method will first try to get the filename from the Content-Disposition header field. If this field does not exist, it then attempts to get the name from the Content-Type header field. If neither field exists, the method will fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AttachFile](#), [GetHeader](#)

CMailMessage::GetBoundary Method

```
LPCTSTR GetBoundary();
```

The **GetBoundary** method returns a pointer to the boundary string used to separate the parts of a multipart message.

Parameters

None.

Return Value

If the method succeeds, the return value is a pointer to the boundary string. If the method fails, a NULL pointer is returned. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreatePart](#), [GetHeader](#), [GetPart](#)

CMailMessage::GetContentDigest Method

```
INT GetContentDigest(  
    LPTSTR lpszDigest,  
    INT cchDigest  
);  
  
INT GetContentDigest(  
    CString& strDigest  
);
```

The **GetContentDigest** method returns an encoded digest of the message.

Parameters

lpszDigest

Pointer to a string buffer to contain the MD5 digest for the specified message. An alternate form of this method accepts a **CString** object which will contain the digest when the method returns.

cchDigest

Maximum length of the digest string, in bytes.

Return Value

If the method succeeds, the return value is the length of the message digest string. A value of zero specifies that there is no MD5 digest for the current message. If the method fails, the return value is `MIME_ERROR`. To get extended error information, call **GetLastError**.

Remarks

This method returns the value of the Content-MD5 header field in the main body of the message. If the header exists, it contains the MD5 digest for the message as defined in RFC 1864.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

CMailMessage::GetContentLength Method

```
LONG GetContentLength();
```

The **GetContentLength** method returns the size of the current message part content in bytes.

Parameters

None.

Return Value

If the method succeeds, the return value is the content length. If the method fails, the return value is `MIME_ERROR`. To get extended error information, call **GetLastError**.

Remarks

This function will return the size of the content in the current message part as the number of bytes and does not account for any Unicode conversion of text. Exercise caution when using this function to determine the size of the buffer that should be allocated for a method like **GetText**. You should always allocate enough memory to accommodate any potential text conversion and decoding which may occur.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHeader](#), [GetPart](#), [GetText](#), [SetPart](#)

CMailMessage::GetDate Method

```
LPCTSTR GetDate(  
    BOOL bLocalize  
);
```

The **GetDate** method returns a pointer to a string that contains the message date and time.

Parameters

bLocalize

Boolean flag which specifies if the date and time should be localized for the current timezone.

Return Value

If the method succeeds, the return value is a pointer to the date and time string. If the method fails, it will return a NULL pointer. To get extended error information, call **GetLastError**.

Remarks

If no date has been specified in the message, the Date header field will be set to the current date and time, and that value will be returned. The date string returned by this method should never be directly modified by the application. Each call to this method will invalidate the previous value that was returned, so if you wish to save or modify the value, you should first make a private copy of the string.

To convert this date string to a long integer value that can be used with the standard C time methods, use the **ParseDate** method. Refer to the **FormatDate** method for information on the format of the date string.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FormatDate](#), [GetHeader](#), [ParseDate](#), [SetDate](#)

CMailMessage::GetErrorString Method

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLastError](#), [SetLastError](#)

CMailMessage::GetExportOptions Method

```
BOOL GetExportOptions(  
    LPDWORD LpdwOptions  
);
```

The **GetExportOptions** method returns a bitmask that describes the default message export options.

Parameters

LpdwOptions

Pointer to mask of attribute options. The mask is a combination of the following values:

Constant	Description
MIME_OPTION_DEFAULT	The default export options. The headers for the message are written out in a specific consistent order, with custom headers written to the end of the header block regardless of the order in which they were set or imported from another message. If the message contains Bcc, Received, Return-Path, Status or X400-Received header fields, they will not be exported.
MIME_OPTION_KEEPOORDER	The original order in which the message header fields were set or imported are preserved when the message is exported.
MIME_OPTION_ALLHEADERS	All headers, including the Bcc, Received, Return-Path, Status and X400-Received header fields will be exported. Normally these headers are not exported because they are only used by the mail transport system. This option can be useful when exporting a message to be stored on the local system, but should not be used when exporting a message to be delivered to another user.
MIME_OPTION_NOHEADERS	When exporting a message, the main header block will not be included. This can be useful when creating a multipart message for services which expect MIME formatted data, such as HTTP POST requests. This option should never be used for email messages being submitted using SMTP.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is 0. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ExportMessage](#), [ImportMessage](#), [SetExportOptions](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CMailMessage::GetFileContentType Method

```
UINT GetFileContentType(  
    LPCTSTR lpszFileName,  
    LPTSTR lpszContentType,  
    INT cchContentType  
);  
  
UINT GetFileContentType(  
    LPCTSTR lpszFileName,  
    CString& strContentType  
);
```

The **GetFileContentType** method returns the content type for the specified file.

Parameters

lpszFileName

Pointer to a string which specifies the name of the file for which content type information is returned.

lpszContentType

Pointer to a string buffer that will contain the MIME type for the specified file. This may be a NULL pointer, in which case this parameter is ignored. If a buffer is provided, it is recommended that it be at least 64 characters in length.

cchContentType

An integer which specifies the maximum number of characters, including the terminating null character, which may be copied into the string buffer.

Return Value

If the method succeeds, the return value is the content type of the specified file. If the method fails, the return value is MIME_CONTENT_UNKNOWN. To get extended error information, call **GetLastError**.

The following values may be returned by this method:

Constant	Description
MIME_CONTENT_UNKNOWN	The file content type is unknown. This value may be returned if the message handle is invalid, or if the file extension is unknown and the file could not be opened for read access.
MIME_CONTENT_APPLICATION	The file content is application specific. Examples of this type of file would be a Microsoft Word document or an executable program. This is also the default type for files which have an unrecognized file name extension and contain binary data.
MIME_CONTENT_AUDIO	The file is an audio file in one of several standard formats. Examples of this type of file would be a Windows (.wav) file or MPEG3 (.mp3) file.
MIME_CONTENT_IMAGE	The file is an image file in one of several standard formats. Examples of this type of file would be a GIF or JPEG image file.
MIME_CONTENT_TEXT	The file is a text file. This is also the default type for files which

	have an unrecognized file name extension and contain only printable text data.
MIME_CONTENT_VIDEO	The file is a video file in one of several standard formats. Examples of this type of file would be a Windows (.avi) or Quicktime (.mov) video file.

Remarks

The content type for a given file is determined based on the file name extension, or if the extension is not recognized, the actual contents of the file. On 32-bit platforms, the system registry is used to determine the default content type values for a given extension. In all cases, file types that are explicitly set using the **SetFileContentType** method will override the default system values.

The content type string which is copied to the string buffer is the standard MIME content type description, which specifies a primary type and a subtype, separated by a slash. For example, a plain text file would have a content type of text/plain, while an HTML document would have a content type of text/html. Binary files may be associated with a specific application. For example, the content type for a Microsoft Word document is application/msword. Those binary files which are not associated with a specific application, or have an unrecognized file name extension, have a content type of application/octet-stream.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AttachData](#), [AttachFile](#), [SetFileContentType](#)

CMailMessage::GetFirstHeader Method

```
BOOL GetFirstHeader(  
    LPTSTR lpszHeader,  
    INT cchMaxHeader,  
    LPTSTR lpszValue,  
    INT cchMaxValue  
);  
  
BOOL GetFirstHeader(  
    CString& strHeader,  
    CString& strValue  
);
```

The **GetFirstHeader** method returns the header field name and value for the first header in the current message part. This method is typically used in conjunction with **GetNextHeader** to enumerate all of the message header fields and their values in the current message part.

Parameters

lpszHeader

A pointer to a string buffer that will contain the the name of the first header in the current message part. This parameter cannot be a NULL pointer.

cchMaxHeader

An integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. This parameter must have a value greater than zero.

lpszValue

A pointer to a string buffer that will contain the value of the first header in the current message part. This parameter may be a NULL pointer, in which case the value of the header field is ignored.

cchMaxValue

An integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. If the *lpszValue* parameter is NULL, this parameter should have a value of zero.

Return Value

If the method succeeds, the return value is non-zero. If no headers exist for the current message part, or the handle to the message is invalid, the method will return zero. To get extended error information, call **GetLastError**.

Remarks

Each part in a multipart message has one or more header fields. To obtain header values for the main message, rather than the message attachments, the current part number must be set to zero using the **SetPart** method.

The header fields from an imported message may not be returned in the same order as which they appear in the message. An application should never make an assumption about the order in which one or more header fields are defined.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnumHeaders](#), [GetPart](#), [GetNextHeader](#), [ParseHeader](#), [SetHeader](#), [SetPart](#)

CMailMessage::GetHandle Method

```
HMESSAGE GetHandle();
```

The **GetHandle** method returns the message handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the message handle associated with the current instance of the class object. If there is no current message, the value `INVALID_MESSAGE` will be returned.

Remarks

This method is used to obtain the message handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

See Also

[AttachHandle](#), [DetachHandle](#), [IsInitialized](#)

CMailMessage::GetHeader Method

```
INT GetHeader(  
    INT nMessagePart,  
    LPCTSTR lpszHeader,  
    LPTSTR lpszValue,  
    INT nMaxValue  
);
```

```
INT GetHeader(  
    LPCTSTR lpszHeader,  
    LPTSTR lpszValue,  
    INT nMaxValue  
);
```

```
INT GetHeader(  
    INT nMessagePart,  
    LPCTSTR lpszHeader,  
    CString& strValue  
);
```

```
INT GetHeader(  
    LPCTSTR lpszHeader,  
    CString& lpszValue  
);
```

The **GetHeader** copies the value of the specified header into a string buffer.

Parameters

nMessagePart

An integer value which specifies which part of the message to return the header value from. A value of zero returns a header value from the main message header, while a value greater than zero returns the header value from that specific part of a multipart message. A value of -1 specifies that the header value should be returned from the current message part. If this argument is omitted, the value will be returned from the current message part.

lpszHeader

A pointer to a string which specifies the message header.

lpszValue

A pointer to a string buffer which will contain the value of the specified header when the method returns. If this parameter is NULL, the method will return the length of the header value without copying the data. This is useful for determining the length of a header value so that a string buffer can be allocated and passed to a subsequent call to the method. In alternate forms of this method, a **CString** object may be specified which will contain the header field value when the method returns.

nMaxValue

An integer value which specifies the maximum number of characters that can be copied to the string buffer, including the terminating null character. If the *lpszValue* parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of bytes copied into the string buffer, not including the terminating null byte. If the *lpszValue* parameter is NULL, the return value is the

length of the header value. If the header field does not exist, a value of zero is returned. If an invalid pointer or message part is specified, a value of MIME_ERROR is returned. To get extended error information, call **GetLastError**.

Example

```
CMailMessage *pMessage = new CMailMessage();
CString strSubject;

// Compose a test message
pMessage->ComposeMessage(_T("Bob Jones <bob@example.com>"),
                        _T("Tom Smith <tom@example.com>"),
                        NULL,
                        _T("This is a test message"),
                        _T("This is a test, this is only a test.));

// Change the subject of the message
if (pMessage->GetHeader(0, _T("Subject"), strSubject))
{
    strSubject = _T("Re: ") + strSubject;
    pMessage->SetHeader(0, _T("Subject"), strSubject);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnumHeaders](#), [GetFirstHeader](#), [GetHeader](#), [GetPart](#), [GetNextHeader](#), [ParseHeader](#), [SetHeader](#), [SetPart](#)

CMailMessage::GetLastError Method

```
DWORD GetLastError();  
  
DWORD GetLastError(  
    CString& strDescription  
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError(0)** when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_MESSAGE or MIME_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

See Also

[GetErrorString](#), [SetLastError](#), [ShowError](#)

CMailMessage::GetMessageSize Method

```
DWORD GetMessageSize(  
    DWORD dwOptions  
);
```

The **GetMessageSize** method returns the size of the complete message in bytes.

Parameters

dwOptions

An optional unsigned integer value which specifies how the size of the message should be calculated, based on what header fields should be included. These are the same options used when exporting a message to a file or memory buffer. The following values may be combined using a bitwise Or operator:

Constant	Description
MIME_OPTION_DEFAULT	The default export options. The headers for the message are written out in a specific consistent order, with custom headers written to the end of the header block regardless of the order in which they were set or imported from another message. If the message contains Bcc, Received, Return-Path, Status or X400-Received header fields, they will not be exported.
MIME_OPTION_ALLHEADERS	All headers, including the Received, Return-Path, Status and X400-Received header fields will be exported. Normally these headers are not exported because they are only used by the mail transport system. This option can be useful when exporting a message to be stored on the local system, but should not be used when exporting a message to be delivered to another user.

Return Value

If the method succeeds, the return value is the size of the current message in bytes. If an error occurs, the method will return `MIME_ERROR`. To get extended error information, call **GetLastError**.

Remarks

This method returns the size of the complete message, including all headers, the message body and any attachments. It can be used to determine the minimum amount of memory that should be allocated to export the message to a memory buffer using the **ExportMessage** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

See Also

[ExportMessage](#), [GetText](#), [ImportMessage](#), [SetExportOptions](#)

CMailMessage::GetNextHeader Method

```
BOOL GetNextHeader(  
    LPTSTR lpszHeader,  
    INT cchMaxHeader,  
    LPTSTR lpszValue,  
    INT cchMaxValue  
);
```

```
BOOL GetNextHeader(  
    CString& strHeader,  
    CString& strValue  
);
```

The **GetNextHeader** method returns the header field name and value for the next header in the current message part. This method is typically used in conjunction with **GetFirstHeader** to enumerate all of the message header fields and their values in the current message part.

Parameters

lpszHeader

A pointer to a string buffer that will contain the name of the next header in the current message part. This parameter cannot be a NULL pointer.

cchMaxHeader

An integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. This parameter must have a value greater than zero.

lpszValue

A pointer to a string buffer that will contain the value of the next header in the current message part. This parameter may be a NULL pointer, in which case the value of the header field is ignored.

cchMaxValue

An integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. If the *lpszValue* parameter is NULL, this parameter should have a value of zero.

Return Value

If the method succeeds, the return value is non-zero. If no more headers exist for the current message part, or the handle to the message is invalid, the method will return zero. To get extended error information, call **GetLastError**.

Remarks

Each part in a multipart message has one or more header fields. To obtain header values for the main message, rather than the message attachments, the current part number must be set to zero using the **SetPart** method.

The header fields from an imported message may not be returned in the same order as which they appear in the message. An application should never make an assumption about the order in which one or more header fields are defined, with the following exception:

If an imported message has multiple Received headers, then those headers will be returned by **GetNextHeader** in the order in which they appeared in the original message. Note that if

GetHeader is used to retrieve the Received header, the first Received header in the message will be returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnumHeaders](#), [GetFirstHeader](#), [GetPart](#), [ParseHeader](#), [SetHeader](#), [SetPart](#)

CMailMessage::GetPart Method

```
INT GetPart();
```

The **GetPart** method returns the current message part index for the specified message.

Parameters

None.

Return Value

If the method succeeds, the return value is the message part index. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

See Also

[CreatePart](#), [DeletePart](#), [GetPartCount](#), [SetPart](#)

CMailMessage::GetPartCount Method

```
INT GetPartCount();
```

The **GetPartCount** method returns the total number of message parts for the specified message. Each message consists of at least one part.

Parameters

None.

Return Value

If the method succeeds, the return value is the number of message parts. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

See Also

[CreatePart](#), [GetPart](#), [SetPart](#)

CMailMessage::GetSender Method

```
INT GetSender(  
    LPCTSTR lpszSender,  
    INT nMaxLength  
);  
  
INT GetSender(  
    CString& strSender  
);
```

The **GetSender** method returns the email address of the message sender in the specified string buffer.

Parameters

lpszSender

A pointer to a string buffer that will contain the email address of the message sender when the method returns. In alternate forms of this method, a **CString** object may be specified which will contain the email address when the method returns.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied to the string buffer, including the terminating null character. It is recommended that the maximum length be at least 64 characters.

Return Value

If the method succeeds, the return value is the number of bytes copied to the string buffer. If an error occurs, the method will return MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method attempts to determine the email address of the sender that originated the message. It will first check for the presence of a Sender or X-Sender header value. If these headers are not defined, it will use the value of the From header field. It will only return successfully if a valid email address can be found.

If the method succeeds, the string buffer that is provided will only contain an email address. It will not contain the display name of the user associated with the address or any extraneous comments that are included in quotes or parenthesis.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnumHeaders](#), [GetFirstHeader](#), [GetHeader](#), [GetPart](#), [GetNextHeader](#), [ParseHeader](#), [SetHeader](#), [SetPart](#)

CMailMessage::GetText Method

```
LONG GetText(  
    LPTSTR lpszBuffer,  
    LONG nMaxLength,  
    LONG nOffset  
);  
  
LONG GetText(  
    CString& strBuffer,  
    LONG nOffset  
);
```

The **GetText** method copies the text of the current message part to the specified buffer.

Parameters

lpszBuffer

A pointer to a string buffer that will receive a copy of the message text when the method returns.

nMaxLength

The maximum number of bytes to copy into the buffer. The size of the buffer provided must be larger than the content length for the current message part.

nOffset

The byte offset from the beginning of the message. A value of zero specifies the first character in the body of the message part.

Return Value

If the method succeeds, the return value is number of bytes copied into the buffer. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

If your project targets a multi-byte character set, this method will always return the message contents as UTF-8 text, regardless of the original character set specified in the message itself. This ensures that characters in the original text are preserved, regardless of the default ANSI code page on the local system. It is recommended you build your project to use Unicode whenever possible. If your application must use ANSI, you can use the **LocalizeText** method to convert the Unicode text to a specific character set.

You should not determine the maximum size of the output buffer using the **GetContentLength** method. That method returns the content size in bytes as it is stored in the message, and does not account for any character encoding or Unicode conversion which may be required. The content length can be used to estimate the amount of text stored in the message part, but you should always allocate a buffer which is larger than the length specified in the message.

If the *nMaxLength* parameter does not specify a buffer size large enough to store the contents of the current message part, this method will fail and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL. Your application must ensure the buffer is large enough to contain the complete message text and a terminating NUL character.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AppendText](#), [ClearText](#), [CompareText](#), [ExportMessage](#), [GetContentLength](#), [ImportMessage](#), [LocalizeText](#), [SetText](#)

CMailMessage::ImportMessage Method

```
BOOL ImportMessage(  
    LPCTSTR lpszFileName  
);
```

```
BOOL ImportMessage(  
    LPCTSTR lpszMessage,  
    INT nLength  
);
```

The **ImportMessage** method imports the message from a file or a string buffer in memory, replacing the current message contents.

Parameters

lpszFileName

A pointer to a string which specifies the name of the file that contains the message to be imported.

lpszMessage

A pointer to a string which contains the message to be imported.

nLength

An integer value which specifies the number of bytes to read from the string buffer. If this argument is -1, then the entire contents of the string up to the terminating null character will be imported.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ExportMessage](#), [ParseBuffer](#)

CMailMessage::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key the initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

See Also

[CMailMessage](#)

CMailMessage::LocalizeText Method

```
LONG LocalizeText(  
    LPCTSTR lpszInput,  
    LONG cchInput,  
    LPBYTE lpOutput,  
    LONG cbOutput,  
    UINT nCharacterSet  
);
```

The **LocalizeText** method converts a Unicode string to its ANSI equivalent using the specified character set.

Parameters

lpszInput

A pointer to a null terminated string which contains the Unicode text which should be localized. If the ANSI version of this function is called, the input text must be in UTF-8 format or the function will fail. This parameter cannot be a NULL pointer.

cchInput

An integer value which specifies the number of characters of text in the input string which should be localized. If this value is -1, the entire length of the string up to the terminating null will be decoded.

lpOutput

A pointer to a byte buffer which will contain the localized ANSI version of the input text. This parameter cannot be a NULL pointer.

cbOutput

An integer value which specifies the maximum number of bytes which can be copied into the output buffer. The buffer must be large enough to store all of the localized text. This value must be greater than zero.

nCharacterSet

An optional numeric identifier which specifies the [character set](#) to use when localizing the text. If this parameter is omitted, the locale for the current thread will be used when localizing the text.

Return Value

If the input text can be successfully localized, the return value is the number of bytes copied into the output buffer. If the function returns zero, then no text was localized. If the function fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **LocalizeText** method enables the application to localize a Unicode string, returning the ANSI version of that text using the specified character set. Because library handles all text as Unicode internally, the ANSI functions in this library will always return UTF-8 encoded text. This method allows you to easily convert that UTF-8 text to another supported character set.

If the ANSI version of this function is called, the input text must use UTF-8 character encoding or the method will fail with the last error set to ST_ERROR_INVALID_CHARACTER_SET.

If the *nCharacterSet* parameter is MIME_CHARSET_DEFAULT or MIME_CHARSET_UNKNOWN the input text will be converted to the default ANSI code page for the current thread locale. If there are characters in the Unicode input text which cannot be converted to an ANSI equivalent using

the specified character set, those characters will be replaced by the default character for your locale, typically a question mark symbol. You cannot specify `MIME_CHARSET_UTF16` as the character set.

This method will always attempt to ensure that the output buffer is terminated with an extra null byte so it is easier to work with as a standard C-style null terminated string. However, if the output buffer is not large enough to accommodate the extra null byte, it will not be copied. It is always recommended that your output buffer be somewhat larger than the length of the original input text to account for multi-byte character sequences. If the output buffer is not large enough to contain the entire localized text, no bytes will be copied to the output buffer and the function will fail with the last error set to `ST_ERROR_BUFFER_TOO_SMALL`.

This function is only required if your application needs to localize the UTF-8 text returned by another function and convert it to a specific 8-bit ANSI character set. For example, if you have an application which calls the ANSI version of **MimeGetMessageText**, it will return the message contents as UTF-8 text. If you need to display that text as localized ANSI, you can call this function to convert the UTF-8 text to your current locale or a specific character set.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeDecodeTextEx](#), [MimeEncodeText](#), [MimeEncodeTextEx](#), [MimeGetMessageText](#),
[MimeSetMessageText](#)

CMailMessage::ParseAddress Method

```
INT ParseAddress(  
    LPCTSTR lpszString,  
    LPCTSTR lpszDomain,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

```
INT ParseAddress(  
    LPCTSTR lpszString,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

```
INT ParseAddress(  
    LPCTSTR lpszString,  
    CString& strAddress  
);
```

The **ParseAddress** method parses a string for an email address, copying the address to the specified buffer.

Parameters

lpszString

A pointer to a string which contains the email address to parse.

lpszDomain

A pointer to a string which specifies a default domain for the address. This parameter may be NULL or point to an empty string if no default domain is required.

lpszAddress

A pointer to a string buffer which will contain the parsed email address when the method returns. It is recommended that this buffer be at least 128 characters in length.

nMaxLength

The maximum number of characters which can be copied into the string buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the address. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **ParseAddress** method is useful for parsing the email addresses that may be specified in various header fields in the message. In many cases, the addresses have additional comment characters which are not part of the address itself. For example, one common format used is as follows:

```
"User Name" <user@domain.com>
```

In this case, the email address is enclosed in angle brackets and the name outside of the brackets is considered to be a comment which is not part of the address itself. Another common format is:

```
user@domain.com (User Name)
```

In this case, there is the address followed by a comment which is enclosed in parenthesis. The

ParseAddress method recognizes both formats, and when passed either string, would return the following address:

user@domain.com

If there was no domain specified in the address, that is just a user name was specified, then the value the *lpzDomain* parameter is added to the address.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ExportMessage](#), [ExtractFile](#), [ImportMessage](#)

CMailMessage::ParseBuffer Method

```
BOOL ParseBuffer(  
    LPCTSTR lpszBuffer,  
    INT cbBuffer  
);
```

The **ParseBuffer** method parses the contents of the specified buffer and adds the contents to the message.

Parameters

lpszBuffer

Pointer to a buffer that contains the text to be added to the message contents.

cbBuffer

The length of the specified buffer. If this value is -1, all characters in the string up to the terminating null character will be parsed.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is useful when the application needs to parse an arbitrary block of text and add it to the specified message. If the buffer contains header fields, the values will be added to the message header. Once the end of the header block is detected, all subsequent text is added to the body of the message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ClearMessage](#), [ImportMessage](#), [ParseHeader](#)

CMailMessage::ParseDate Method

```
BOOL ParseDate(  
    LPCTSTR lpszDate,  
    LPLONG lpnSeconds,  
    LPLONG lpnTimezone,  
    BOOL bLocalize  
);
```

The **ParseDate** method parses a date string, returning the number of seconds since 1 January 1970 and the difference in seconds between the specified timezone and coordinated universal time. If the string does not specify a timezone, the local timezone is used.

Parameters

lpszDate

A pointer to a string which specifies the date to be parsed. The date string must be in the standard format defined by RFC 822.

lpnSeconds

A pointer to a long integer which is set to the number of seconds since 1 January 1970 00:00:00 UTC. This date is commonly called the epoch, and is the base date used by the standard C time methods. This pointer may be NULL, in which case the parameter is ignored.

lpnTimezone

A pointer to a long integer which is set to the difference in seconds between the specified date's timezone and coordinated universal time (also known as Greenwich Mean Time). This pointer may be NULL, in which case the parameter is ignored.

bLocalize

A boolean flag which determines if the date should be localized to the current timezone, regardless of the timezone specified in the date string. A non-zero value specifies the timezone for the local system will be used, adjusted for daylight savings time if applicable.

Return Value

If the date could be successfully parsed, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is not a general purpose date parsing function, and may not be capable of parsing dates for a specific locale. The date and time should be in a standard format as outlined in RFC 822, which describes the basic structure of Internet email messages. For a description of the date string format, refer to the **MimeFormatDate** function.

If the date and time does not include any timezone information, Coordinated Universal Time (UTC) will be used by default. This is an important consideration if you use this function to parse input from a user, because in most cases they will not provide a timezone and will assume the date and time they enter is for their current timezone.

The value of the *bLocalize* parameter will only change the number of seconds offset by the current timezone and does not affect the value returned in the *lpnSeconds* parameter. If the date is localized, the timezone offset will be adjusted for daylight savings if it was in effect at the time.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FormatDate](#), [GetDate](#), [SetDate](#)

CMailMessage::ParseHeader Method

```
BOOL ParseHeader(  
    LPCTSTR lpszBuffer  
);
```

The **ParseHeader** method parses a line of text and adds the header and value to the current message.

Parameters

lpszBuffer

Pointer to a string which contains the header and value to be added to the message.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to parse a line of text that is part of a message header. The string must consist of a header name, followed by a colon, followed by the header value. The header name may only consist of printable characters, and may not contain whitespace (space, tab, carriage return or linefeed characters).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImportMessage](#), [ParseBuffer](#)

CMailMessage::SetDate Method

```
BOOL SetDate(  
    LPCTSTR lpszDate,  
    BOOL bLocalize  
);
```

The **SetDate** method sets the date and time in the header for the specified message.

Parameters

lpszDate

Pointer to a string which specifies the date and time. If this parameter specifies a zero-length string or a NULL pointer, the current date and time will be used.

bLocalize

Boolean flag that specifies the date and time should be localized for the current timezone.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The date string should be in a standard format as outlined in RFC 822, the document which describes the basic structure of Internet email messages. For a description of the date string format, refer to the **FormatDate** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FormatDate](#), [GetDate](#), [GetHeader](#), [ParseDate](#), [SetHeader](#)

CMailMessage::SetExportOptions Method

```
BOOL SetExportOptions(  
    DWORD dwOptions  
);
```

The **SetExportOptions** method specifies a bitmask that describes current message export options.

Parameters

dwOptions

Mask of attribute options. The mask is a combination of the following values:

Constant	Description
MIME_OPTION_DEFAULT	The default export options. The headers for the message are written out in a specific consistent order, with custom headers written to the end of the header block regardless of the order in which they were set or imported from another message. If the message contains Bcc, Received, Return-Path, Status or X400-Received header fields, they will not be exported.
MIME_OPTION_KEEPOORDER	The original order in which the message header fields were set or imported are preserved when the message is exported.
MIME_OPTION_ALLHEADERS	All headers, including the Bcc, Received, Return-Path, Status and X400-Received header fields will be exported. Normally these headers are not exported because they are only used by the mail transport system. This option can be useful when exporting a message to be stored on the local system, but should not be used when exporting a message to be delivered to another user.
MIME_OPTION_NOHEADERS	When exporting a message, the main header block will not be included. This can be useful when creating a multipart message for services which expect MIME formatted data, such as HTTP POST requests. This option should never be used for email messages being submitted using SMTP.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

By default, the Received and Return-Path headers are not exported. In addition, the order of the headers in an exported message is undefined. This is reasonable behavior for most mail clients, but may not be appropriate for applications which need access to all of the header fields.

Example

```
pMessage->SetExportOptions(MIME_EXPORT_OPTIONS_ALL |
                           MIME_EXPORT_OPTIONS_KEEP_ORDER);

if (pMessage->Import(lpszFileName))
{
    // Process the message and make any modifications
    // then write the message back out, preserving all
    // of the headers in their original order
    bResult = pMessage->Export(lpszFileName);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ExportMessage](#), [ImportMessage](#), [GetExportOptions](#)

CMailMessage::SetFileContentType Method

```
BOOL SetFileContentType(  
    UINT nContentType,  
    LPCTSTR lpszSubtype,  
    LPCTSTR lpszExtension  
);
```

The **SetFileContentType** method associates a per-message content type with a given file name extension. This association is specific to the message, and is not shared by other messages that may be opened by the process.

Parameters

nContentType

An identifier which specifies the primary content type. It may be one of the following values:

Constant	Description
MIME_CONTENT_UNKNOWN	The default content type for the specified extension should be used. This value should only be used to delete a previously defined content type.
MIME_CONTENT_APPLICATION	The file content is application specific. Examples of this type of file would be a Microsoft Word document or an executable program. This is also the default type for files which have an unrecognized file name extension and contain binary data.
MIME_CONTENT_AUDIO	The file is an audio file in one of several standard formats. Examples of this type of file would be a Windows (.wav) file or MPEG3 (.mp3) file.
MIME_CONTENT_IMAGE	The file is an image file in one of several standard formats. Examples of this type of file would be a GIF or JPEG image file.
MIME_CONTENT_TEXT	The file is a text file. This is also the default type for files which have an unrecognized file name extension and contain only printable text data.
MIME_CONTENT_VIDEO	The file is a video file in one of several standard formats. Examples of this type of file would be a Windows (.avi) or Quicktime (.mov) video file.

lpszSubtype

A pointer to a string which specifies the MIME subtype. This parameter may be NULL if the content type association is being deleted.

lpszExtension

A pointer to a string which specifies the file name extension that will be associated with the MIME content type.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetFileContentType** method allows an application to specify a content type for a given file extension, and is typically used to define custom content types for file attachments. The content type will override any default content types associated with the extension, as well as allow new content types to be defined for application-specific files.

Example

In the following example, the file extension ".dat" is associated with a custom content type, a file is attached to the message and then the custom content type is deleted. Note that because the primary content type designates the file as an application specific (non-text) file, it will be automatically encoded when attached to a message:

```
bResult = pMessage->SetFileContentType(
    MIME_CONTENT_APPLICATION,
    _T("octet-stream"), _T("dat"));

if (bResult)
{
    bResult = pMessage->AttachFile(lpszFileName, MIME_ATTACH_DEFAULT);
    pMessage->SetFileContentType(MIME_CONTENT_UNKNOWN, NULL, _T("dat"));
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AttachData](#), [AttachFile](#), [GetFileContentType](#)

CMailMessage::SetHeader Method

```
BOOL SetHeader(  
    INT nMessagePart,  
    LPCTSTR lpszHeader,  
    LPCTSTR lpszValue  
);
```

```
BOOL SetHeader(  
    LPCTSTR lpszHeader,  
    LPCTSTR lpszValue  
);
```

The **SetHeader** method adds or updates a header field in the specified message.

Parameters

nMessagePart

An integer value which specifies which part of the message the header should be set or modified in. A value of zero sets a header value in the main message header block, while a value greater than zero sets the header value in that specific part of a multipart message. If this argument is omitted or a value of -1 is specified, the header value will be set in the current message part.

lpszHeader

Pointer to a string which specifies the header field that will be added or updated.

lpszValue

Pointer to a string which specifies the value for the header field. This pointer may be NULL, which causes the header field to be removed from the message.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnumHeaders](#), [GetFirstHeader](#), [GetHeader](#), [GetPart](#), [GetNextHeader](#), [SetPart](#), [SetText](#)

CMailMessage::SetLastError Method

```
VOID SetLastError(  
    DWORD dwErrorCode  
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_MESSAGE or MIME_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

See Also

[GetErrorString](#), [GetLastError](#), [ShowError](#)

CMailMessage::SetPart Method

```
INT SetPart(  
    INT nNewPart  
);
```

The **SetPart** method sets the current message part index for the specified message.

Parameters

nNewPart

The new message part index. A value of zero specifies the main message part.

Return Value

If the method succeeds, the return value is the previous message part index. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetPart](#), [GetPartCount](#), [SetHeader](#)

CMailMessage::SetText Method

```
LONG SetText(  
    LONG nOffset,  
    LPCTSTR lpszText  
);  
  
LONG SetText(  
    LPCTSTR lpszText  
);
```

The **SetText** method copies the specified text into the body of the current message part.

Parameters

nOffset

The offset into the body of the message part. A value of -1 specifies that the text will be appended to the message body. If this argument is omitted, the current message text is replaced with the contents of the *lpszText* argument.

lpszText

A pointer to a string which specifies the text to be copied to the current message part at the given offset.

Return Value

If the method succeeds, the return value is the number of bytes copied into the message. A return value of zero indicates that no text could be copied into the current message part. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AppendText](#), [ClearMessage](#), [ClearText](#), [CompareText](#), [GetText](#), [SetHeader](#), [SetPart](#)

CMailMessage::ShowError Method

```
INT ShowError(  
    LPCTSTR lpszAppTitle,  
    UINT uType,  
    DWORD dwErrorCode  
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A pointer to a string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetErrorString](#), [GetLastError](#), [SetLastError](#)

CMailMessage Character Sets

Constant	Value	Name	Code Page	Description
MIME_CHARSET_USASCII	1	us-ascii	20127	A character set which defines 7-bit printable characters with values ranging from 20h to 7Eh. An application that uses this character set has the broadest compatibility with most mail servers (MTAs) because it does not require the server to handle 8-bit characters correctly when the message is delivered.
MIME_CHARSET_ISO8859_1	2	iso-8859-1	28591	A character set for most western European languages such as English, French, Spanish and German. This character set is also commonly referred to as Latin-1. This character set is similar to Windows code page 1252 (Windows-1252), however there are differences such as the Euro symbol.
MIME_CHARSET_ISO8859_2	3	iso-8859-2	28592	A character set for most central and eastern European languages such as Czech, Hungarian, Polish and Romanian. This character set is also commonly referred to as Latin-2. This character set is similar to Windows code page 1250, however the characters are arranged differently.
MIME_CHARSET_ISO8859_3	12	iso-8859-3	28593	A character set for southern European languages such as Maltese and Esperanto. This character set was also used with the Turkish language, but it was superseded by ISO 8859-9 which is the preferred character set for Turkish. This character set is not widely used in mail messages and it is recommended that you use UTF-8 instead.
MIME_CHARSET_ISO8859_4	13	iso-8859-4	28594	A character set for northern European languages such as

				Latvian, Lithuanian and Greenlandic. This character set is not widely used in mail messages and it is recommended that you use UTF-8 instead.
MIME_CHARSET_ISO8859_5	4	iso-8859-5	28595	A character set for Cyrillic languages such as Russian, Bulgarian and Serbian. This character set was never widely adopted and most mail messages use either KOI8 or UTF-8 encoding.
MIME_CHARSET_ISO8859_6	5	iso-8859-6	28596	A character set for Arabic languages. Note that the application is responsible for displaying text that uses this character set. In particular, any display engine needs to be able to handle the reverse writing direction and analyze the context of the message to correctly combine the glyphs.
MIME_CHARSET_ISO8859_7	6	iso-8859-7	28597	A character set for the Greek language. This character set is also commonly referred to as Latin/Greek. This character set is no longer widely used and has largely been replaced with UTF-8 which provides more complete coverage of the Greek alphabet.
MIME_CHARSET_ISO8859_8	7	iso-8859-8	28598	A character set for the Hebrew language. Note that similar to Arabic, Hebrew uses a reverse writing direction. An application which displays this character should be capable of processing bi-directional text where a single message may include both right-to-left and left-to-right languages, such as Hebrew and English. In most cases it is recommended that you use UTF-8 instead of this character set.
MIME_CHARSET_ISO8859_9	8	iso-8859-9	28599	A character set for the Turkish language. This character set is

				also commonly referred to as Latin-5. This character set is nearly identical to ISO 8859-1, except that it replaces certain Icelandic characters with Turkish characters.
MIME_CHARSET_ISO8859_10	14	iso-8859-10	28600	A character set for the Danish, Icelandic, Norwegian and Swedish languages. This character set is also commonly referred to as Latin-6 and is similar to ISO 8859-4.
MIME_CHARSET_ISO8859_13	15	iso-8859-13	28603	A character set for Baltic languages. This character set is also commonly referred to as Latin-7. This character set is similar to ISO 8859-4, except it adds certain Polish characters and does not support Nordic languages.
MIME_CHARSET_ISO8859_14	16	iso-8859-14	28604	A character set for Gaelic languages such as Irish, Manx and Scottish Gaelic. This character set is also commonly referred to as Latin-8. This character set replaced ISO 8859-12 which was never fully implemented.
MIME_CHARSET_ISO8859_15	17	iso-8859-15	28605	A character set for western European languages. This character set is also commonly referred to as Latin-9 and is nearly identical to ISO8859-1 except that it replaces lesser-used symbols with the Euro sign and some letters.
MIME_CHARSET_ISO2022_JP	18	iso-2022-jp	50222	A multi-byte character encoding for Japanese that is widely used with mail messages. This is a 7-bit encoding where all characters start with ASCII and uses escape sequences to switch to the double-byte character sets.
MIME_CHARSET_ISO2022_KR	19	iso-2022-kr	50225	A multi-byte character encoding for Korean which encodes both ASCII and Korean

				double-byte characters. This is a 7-bit encoding which uses the shift in and shift out control characters to switch to the double-byte character set.
MIME_CHARSET_ISO2022_CN	20	x-cp50227	50227	A multi-byte character encoding for Simplified Chinese which encodes both ASCII and Chinese double-byte characters. This is a 7-bit encoding which uses the shift in and shift out control characters to switch to the double-byte character set.
MIME_CHARSET_KOI8R	21	koi8-r	20866	A character set for Russian using the Cyrillic alphabet. This character set also covers the Bulgarian language. Most mail messages in the Russian language use this character set or UTF-8 instead of ISO 8859-5, which was never widely adopted.
MIME_CHARSET_KOI8U	22	koi8-u	21866	A character set for Ukrainian using the Cyrillic alphabet. This character set is similar to the KOI8-R character set, but replaces certain symbols with Ukrainian letters. Most mail messages in the Ukrainian language use this character set or UTF-8 instead of ISO 8859-5, which was never widely adopted.
MIME_CHARSET_GB2312	23	x-cp20936	20936	A multi-byte character encoding which can represent ASCII and simplified Chinese characters. It has been superseded by GB18030, however it remains widely used in China.
MIME_CHARSET_GB18030	24	gb18030	54936	A Unicode transformation format which can represent all Unicode code points and supports both simplified and traditional Chinese characters. It is backwards compatible with GB2312 and supersedes that

				character set.
MIME_CHARSET_BIG5	25	big5	950	A multi-byte character set that supports both ASCII characters and traditional Chinese characters. It is widely used in Taiwan, Hong Kong and Macau. It is no longer commonly used in China, which has developed GB18030 as a standard encoding. Microsoft's implementation of Big5 on Windows does not support all of the extensions and is missing certain code points.
MIME_CHARSET_UTF7	9	utf-7	65000	A Unicode transformation format that uses variable-length character encoding to represent Unicode text as a stream of ASCII characters that are safe to transport between mail servers that only support 7-bit printable characters. It is primarily used as an alternative to UTF-8 when quoted-printable or base64 encoding is not desired.
MIME_CHARSET_UTF8	10	utf-8	65001	A Unicode transformation format that uses multi-byte character sequences to represent Unicode text. It is backwards compatible with the ASCII character set, however because it uses 8-bit text, it is recommended that you use either quoted-printable or base64 encoding to ensure compatibility with mail servers that do not support 8-bit characters.
MIME_CHARSET_UTF16	11	utf-16le	N/A	A 16-bit Unicode format that represents each character as a 16-bit value in little endian byte order. This character set is not widely used in mail messages and it is recommended that you use UTF-8 instead. UTF-16 characters in big endian byte order are not supported.

Remarks

When composing a new message, it is recommended that you always use UTF-8 as the character set encoding which ensures broad compatibility with most applications. The other character sets are primarily used when parsing messages generated by other applications. Internally, all message headers and text are processed as UTF-8. If you compile your application using a multi-byte (ANSI) character set, header values and message text will always be returned to your application as UTF-8 encoded Unicode, regardless of the original character set used in the message.

In addition to the character sets listed above, the class will recognize additional character sets which correspond to specific Windows code pages, as well several variants. These additional character sets are included for compatibility with other applications; they are not defined because they should not be used when composing new messages.

It is important to note that while certain Windows character sets are similar to standard ISO character sets, they are not identical. For example, although the Windows-1252 character set is nearly identical to ISO 8859-1, they are not interchangeable. Some legacy applications make the error of representing Windows ANSI character sets as 8-bit ISO character sets, which can result in errors when converting them to Unicode. This is something to be aware of when encoding and decoding text generated by older applications. Before the widespread adoption of UTF-8, it was particularly common for legacy Windows mail clients to default to using Windows-1252 for text and label it as using ISO 8859-1.

Although the **CMailMessage** class supports UTF-16, it is recommended you use UTF-8 instead. Text which uses UTF-16 will always be base64 encoded, and some mail clients may not recognize it as a valid character set. If the message does not specify if big endian or little endian byte order is used, the class will default to little endian. When UTF-16 is used when composing a new message, it will always use little endian byte order.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

See Also

[CMailMessage](#), [ComposeMessage](#), [CreatePart](#), [DecodeText](#), [EncodeText](#)

Message Store Class Library

Store a collection of email messages on the local system.

Reference

- [Class Methods](#)
- [Error Codes](#)

Library Information

Class Name	CMessageStore
File Name	CSMSGV10.DLL
Version	10.0.1468.2518
LibID	E6D79220-7873-4170-BF39-5D7A0049655C
Import Library	CSMSGV10.LIB
Dependencies	None

Overview

The Message Store class library provides an interface for managing a local message storage file that can be used to store and retrieve multiple messages. Functions are provided to open and create storage files, add, remove and extract messages from storage, and search the stored messages for specific header field values.

SocketTools also includes a class library for managing individual messages. For more information, refer to the [CMailMessage](#) class.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Message Store Class Methods

Class	Description
CMessageStore	Constructor which initializes the current instance of the class
~CMessageStore	Destructor which releases resources allocated by the class
Method	Description
AttachHandle	Attach the specified message handle to this instance of the class
CloseFile	Close the current message storage file
CopyFile	Duplicate the contents of the current message store in a new file
DeleteMessage	Remove the specified message from the current message store
DetachHandle	Detach the handle for the current instance of this class
FindMessage	Search for a message in the current message store
GetHandle	Return the message handle used by this instance of the class
GetErrorString	Return a description for the specified error code
GetLastError	Return the last error code
GetMessage	Retrieve a message from the current message store
GetMessageCount	Return the number of messages in the current message store
IsInitialized	Determine if the class has been successfully initialized
OpenFile	Open the specified message storage file
PurgeFile	Purge all deleted messages from the current message store
ReplaceMessage	Replace the specified message in the current message store
SetLastError	Set the last error code
ShowError	Display a message box with a description of the specified error
StoreMessage	Store the specified message in the current message store

CMessageStore::CMessageStore Method

`CMessageStore();`

The **CMessageStore** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

See Also

[~CMessageStore](#), [IsInitialized](#)

CMessageStore::~CMessageStore

`~CMessageStore();`

The **CMessageStore** destructor releases resources allocated by the current instance of the **CMessageStore** object. It also uninitialized the library if there are no other concurrent uses of the class.

Remarks

When a **CMessageStore** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. All handles that were created for the session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

See Also

[CMessageStore](#)

CMessageStore::AttachHandle Method

```
VOID AttachHandle(  
    HMESSAGESTORE hStorage  
);
```

The **AttachHandle** method attaches the specified message store to the current instance of the class.

Parameters

hStorage

The handle to the message store that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a message store handle created outside of the class using the SocketTools API. Once the handle is attached to the class, the other class member functions may be used with that message store.

If a message store handle already has been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous message store from being closed, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the *hStorage* parameter is presumed to be a valid message store handle and no checks are performed to ensure that the handle is valid. Specifying an invalid message store handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

See Also

[DetachHandle](#), [GetHandle](#)

CMessageStore::CloseFile Method

```
BOOL CloseFile(  
    BOOL bPurgeMessages  
);
```

The **CloseFile** method closes the current message store.

Parameters

bPurgeMessages

An integer value which specifies if deleted messages are purged from the message store. A non-zero value specifies that all messages marked for deletion will be removed from the message store. A value of zero specifies that deleted messages will not be removed from the store.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **CloseFile** method closes the storage file that was previously opened, releasing all of the memory allocated for the message store and optionally purging all deleted messages. Note that the current message store will be closed by the class destructor when the class object is deleted or goes out of scope.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

See Also

[CopyFile](#), [OpenFile](#), [PurgeFile](#)

CMessageStore::CopyFile Method

```
BOOL CopyFile(  
    LPCTSTR lpszFileName  
);
```

The **CopyFile** method duplicates the contents of the current message store in a new file.

Parameters

lpszFileName

A pointer to a string which specifies the name of the file that the messages will be copied to. This parameter cannot be NULL and must specify a valid file path and name.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The **CopyFile** method is used to create a copy of the current message store in a new file. If the file does not exist, it will be created. If the file already exists, then the contents will be overwritten with the contents of the message store.

Messages that have been marked for deletion are not copied to the new message store file.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CloseFile](#), [OpenFile](#), [PurgeFile](#)

CMessageStore::DeleteMessage Method

```
BOOL DeleteMessage(  
    LONG nMessageId  
);
```

The **DeleteMessage** method removes the specified message from the current message store.

Parameters

nMessageId

An integer value which identifies the message that is to be removed from the message store. Message numbers begin at one and increment for each message in the store.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **DeleteMessage** method marks the specified message for deletion from the storage file. When the message store is closed or purged, the message is removed from the file. Once a message has been marked for deletion, it may no longer be referenced by the application. For example, you cannot access the contents of a message that has been deleted.

The message store must be opened with write access. This method will fail if you attempt to delete a message from a storage file that has been opened for read-only access. If the application needs to delete messages in the message store, it is recommended that the file be opened for exclusive access using the MIME_STORAGE_LOCK option.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

See Also

[CloseFile](#), [FindMessage](#), [GetMessage](#), [GetMessageCount](#), [PurgeFile](#)

CMessageStore::DetachHandle Method

```
HMESSAGESTORE DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the message store handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_MESSAGESTORE` will be returned.

Remarks

This method is used to detach a message handle created by the class for use with the SocketTools API. Once the message handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

See Also

[AttachHandle](#), [GetHandle](#)

CMessageStore::FindMessage Method

```
LONG FindMessage(  
    LONG nMessageId,  
    LPCTSTR lpszHeaderName,  
    LPCTSTR lpszHeaderValue,  
    DWORD dwOptions  
);
```

The **FindMessage** method searches for a message in the current message store.

Parameters

nMessageId

An integer value which specifies the message number that should be used when starting the search. The first message in the message store has a value of one.

lpszHeaderField

A pointer to the string which specifies the name of the header field that should be searched. The header field name is not case sensitive. This parameter cannot be NULL.

lpszHeaderValue

A pointer to the string which specifies the header value that should be searched for. The search options can be used to specify if the search is case-sensitive, and whether the search should return partial matches to the string. This parameter cannot be NULL.

dwOptions

A value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
MIME_SEARCH_DEFAULT	Perform a complete match against the specified header value. The comparison is not case-sensitive.
MIME_SEARCH_CASE_SENSITIVE	The header field value comparison will be case-sensitive. Note that this does not affect header field names. Matches for header names are always case-insensitive.
MIME_SEARCH_PARTIAL_MATCH	Perform a partial match against the specified header value. It is recommended that this option be used when searching for matches to email addresses.
MIME_SEARCH_DECODE_HEADERS	Decode any encoded message headers before comparing them to the specified value. This option can increase the amount of time required to search the message store and should only be used when necessary.

Return Value

If the method succeeds, the return value is the number for the message which matches the search criteria. If the method fails, the return value is `MIME_ERROR`. To get extended error information,

call `GetLastError`.

Remarks

The **FindMessage** method is used to search the message store for a message which matches a specific header field value. For example, it can be used to find every message which is addressed to a specific recipient or has a subject which matches a particular string value.

Example

```
CMessageStore mailStorage;
LPCTSTR lpszHeader = _T("From");
LPCTSTR lpszAddress = _T("jsmith@example.com");
LONG nMessageId = 1;

if (! mailStorage.OpenFile(lpszFileName))
{
    // Unable to open the storage file
    return;
}

// Begin searching for messages from the specified sender
while (nMessageId != MIME_ERROR)
{
    nMessageId = mailStorage.FindMessage(nMessageId,
                                        lpszHeader,
                                        lpszAddress,
                                        MIME_SEARCH_PARTIAL_MATCH);

    if (nMessageId != MIME_ERROR)
    {
        CMailMessage mailMessage;

        // Get a copy of the message that was found
        if (mailStorage.GetMessage(nMessageId, mailMessage))
        {
            // Store the message in a file
            TCHAR szFileName[MAX_PATH];
            BOOL bExported;

            // Create a filename based on the message number
            wsprintf(szFileName, _T("msg%05ld.tmp"), nMessageId);

            // Export the message to a file
            bExported = mailMessage.ExportMessage(szFileName);
        }

        // Increase the message ID to resume the search at the next message
        nMessageId++;
    }
}

mailStorage.CloseFile();
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DeleteMessage](#), [GetMessage](#), [GetMessageCount](#)

CMessageStore::GetErrorString Method

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLastError](#), [SetLastError](#)

CMessageStore::GetHandle Method

```
HMESSAGESTORE GetHandle();
```

The **GetHandle** method returns the message store handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the message store handle associated with the current instance of the class object. If there is no current message, the value `INVALID_MESSAGESTORE` will be returned.

Remarks

This method is used to obtain the message store handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

See Also

[AttachHandle](#), [DetachHandle](#), [IsInitialized](#)

CMessageStore::GetLastError Method

```
DWORD GetLastError();  
  
DWORD GetLastError(  
    CString& strDescription  
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError(0)** when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_MESSAGESTORE or MIME_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

See Also

[GetErrorString](#), [SetLastError](#), [ShowError](#)

CMessageStore::GetMessage Method

```
BOOL GetMessage(  
    LONG nMessageId,  
    CMailMessage& mailMessage  
);  
  
HMESSAGE GetMessage(  
    LONG nMessageId,  
    DWORD dwOptions  
);
```

The **GetMessage** method retrieves a message from the current message store.

Parameters

nMessageId

An integer value which specifies the message number that should be retrieved. The first message in the message store has a value of one.

mailMessage

A **CMailMessage** object which will reference the message that is retrieved from the message store.

dwOptions

A value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
0	A shared message handle returned by the function. The contents of this message will be overwritten each time this function is called.
MIME_COPY_STORED_MESSAGE	A message handle is allocated for a copy of the message that is retrieved from the message store.

Return Value

If the first form of the method succeeds the return value is non-zero, otherwise the return value is zero. If the second form of the message succeeds it returns a message handle, otherwise it returns **INVALID_MESSAGE**. To get extended error information, call **GetLastError**.

Remarks

The second form of the **GetMessage** method returns a message handle for the specified message in the message store. If no options are specified, a temporary message handle is returned that is only valid until the next message is retrieved. If a multithreaded application changes the contents of the temporary message, it will change for all other threads that have obtained a message handle using this function.

If the application must have a unique copy of the message, the **MIME_COPY_STORED_MESSAGE** option should be specified. Instead of returning a handle to a shared message, the message is duplicated and a handle to that copy of the message is returned. If a reference to a **CMailMessage** object is passed to the first form of this method, this option is used to create a copy of the message.

Example

```
CMessageStore mailStorage;
LPCTSTR lpszHeader = _T("From");
LPCTSTR lpszAddress = _T("jsmith@example.com");
LONG nMessageId = 1;

if (! mailStorage.OpenFile(lpszFileName))
{
    // Unable to open the storage file
    return;
}

// Begin searching for messages from the specified sender
while (nMessageId != MIME_ERROR)
{
    nMessageId = mailStorage.FindMessage(nMessageId,
                                        lpszHeader,
                                        lpszAddress,
                                        MIME_SEARCH_PARTIAL_MATCH);

    if (nMessageId != MIME_ERROR)
    {
        CMailMessage mailMessage;

        // Get a copy of the message that was found
        if (mailStorage.GetMessage(nMessageId, mailMessage))
        {
            // Store the message in a file
            TCHAR szFileName[MAX_PATH];
            BOOL bExported;

            // Create a filename based on the message number
            wsprintf(szFileName, _T("msg%05ld.tmp"), nMessageId);

            // Export the message to a file
            bExported = mailMessage.ExportMessage(szFileName);
        }

        // Increase the message ID to resume the search at the next message
        nMessageId++;
    }
}

mailStorage.CloseFile();
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

See Also

[CMailMessage](#), [DeleteMessage](#), [FindMessage](#), [GetMessageCount](#), [StoreMessage](#)

CMessageStore::GetMessageCount Method

```
BOOL GetMessageCount(  
    LPLONG lpnLastMessage  
);
```

The **GetMessageCount** method returns the number of messages in the current message store.

Parameters

lpnLastMessage

A pointer to an integer which will contain the message number for the last message in the storage file. If this information is not required, a NULL pointer may be specified.

Return Value

If the method succeeds, the return value is the number of messages in the message store. If the method fails, the return value is MIME_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessageCount** method returns the number of messages in the message store. It is important to note that does not count those messages which have been marked for deletion. This means that the value returned by this function will decrease as messages are deleted.

The message number returned in the *lpnLastMessage* parameter will specify the total number of messages in the message store, including deleted messages.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DeleteMessage](#), [FindMessage](#), [GetMessage](#)

CMessageStore::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key the initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

See Also

[CMessageStore](#)

CMessageStore::OpenFile Method

```
BOOL OpenFile(  
    LPCTSTR lpszFileName,  
    DWORD dwOpenMode  
);
```

The **OpenFile** method opens the specified message storage file.

Parameters

lpszFileName

A pointer to a string which specifies the name of the storage file.

dwOpenMode

A value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
MIME_STORAGE_READ	The message store will be opened for read access. The contents of the message store can be accessed, but cannot be modified by the process unless it has also been opened for writing.
MIME_STORAGE_WRITE	The message store will be opened for writing. This mode also implies read access and must be specified if the application needs to modify the contents of the message store.
MIME_STORAGE_CREATE	The message store will be created if the storage file does not exist. If the file exists, it will be truncated. This mode implies read and write access.
MIME_STORAGE_LOCK	The message store will be opened so that it may only be accessed and modified by the current process.
MIME_STORAGE_COMPRESS	The contents of the message store are compressed. This option is automatically enabled if a compressed message store is opened for reading or writing.
MIME_STORAGE_MAILBOX	The message store should use the UNIX mbox format when reading and storing messages. This option is provided for backwards compatibility and is not recommended for general use.

Return Value

If the method succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **OpenFile** method opens a message storage file which contains one or more messages. If the storage file is opened for read access, the application can search the message store and extract messages but it cannot add or delete messages. To add new messages or delete existing messages from the store, it must be opened with write access.

The message store is designed to be a simple, effective way to store multiple messages together in a single file. When the message store is opened, the contents are indexed in memory. Although there is no specific limit to the number of messages that can be stored, there must be sufficient memory available to build an index of each message and its headers. If the application must store and manage a very large number of messages, it is recommended that you use a database rather than a flat-file message store.

Message Store Format

Each message is prefixed by a control sequence of five ASCII 01 characters followed by an ASCII 10 and ASCII 13 character. The messages themselves are stored unmodified in their original text format. The length of each message is calculated based on the location of the control sequence that delimits each message, and explicit message lengths are not stored in the file. This means that it is safe to manually change the message contents, as long as the message delimiters are preserved.

If the message store is compressed, the contents of the storage file are expanded when the file is opened and then re-compressed when the storage file is closed. Using the `MIME_STORAGE_COMPRESS` option reduces the size of the storage file and prevents the contents of the message store from being read using a text file editor. However, enabling compression will increase the amount of memory allocated by the library and can increase the amount of time that it takes to open and close the storage file.

The class also has a backwards compatibility mode where it will recognize storage files that use the UNIX mbox format. While this format is supported for accessing existing files, it is not recommended that you use it when creating new message stores or adding messages to an existing store. There are a number of different variants on the mbox format that have been used by different Mail Transfer Agents (MTAs) on the UNIX platform. For example, the `mboxrd` variant looks identical to the `mboxcl2` variant, and they are programmatically indistinguishable from one another, but they are not compatible. For this reason, the use of the mbox format is strongly discouraged.

Example

```
CMailMessage mailMessage;

// Compose a new message
mailMessage.ComposeMessage(lpszSender,
                           lpszRecipient,
                           NULL,
                           lpszSubject,
                           lpszMessage,
                           NULL,
                           MIME_CHARSET_DEFAULT,
                           MIME_ENCODING_DEFAULT);

CMessageStore mailStorage;

// Open the message storage file
if (mailStorage.OpenFile(lpszFileName, MIME_STORAGE_WRITE))
{
    // Store a copy of the message in the message store
    nMessageId = mailStorage.StoreMessage(mailMessage);

    if (nMessageId == MIME_ERROR)
    {
        // We were unable to store the message
    }
}
```

```
    }  
  
    // Close the message store  
    mailStorage.CloseFile();  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CloseFile](#), [FindMessage](#), [GetMessage](#), [GetMessageCount](#), [PurgeFile](#)

CMessageStore::PurgeFile Method

```
BOOL PurgeFile();
```

The **PurgeFile** method purges all deleted messages from the specified message store.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **PurgeFile** method purges all deleted messages from the message store. If the storage file has been opened in read-only mode or there are no messages marked for deletion, this method will take no action.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

See Also

[CopyFile](#), [DeleteMessage](#), [FindMessage](#), [GetMessage](#)

CMessageStore::ReplaceMessage Method

```
BOOL ReplaceMessage(  
    LONG nMessageId,  
    CMailMessage& mailMessage  
);
```

```
BOOL ReplaceMessage(  
    LONG nMessageId,  
    HMESSAGE hMessage  
);
```

The **MimeReplaceStoredMessage** function replaces the contents of the specified message in a message store.

Parameters

nMessageId

An integer value which specifies the message number that should be replaced.

mailMessage

A **CMailMessage** object which references the message that will be stored.

hMessage

Handle to the message that will be stored.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The **ReplaceMessage** method replaces the specified message with a new message. The message number may be a message that has been previously marked for deletion. It is important to note that the change will not be reflected in the physical storage file until it has been closed. If the application needs to replace messages in the message store, it is recommended that the file be opened for exclusive access using the MIME_STORAGE_LOCK option.

Example

```
CMailMessage mailMessage;  
  
// Compose a new message  
mailMessage.ComposeMessage(lpszSender,  
                           lpszRecipient,  
                           NULL,  
                           lpszSubject,  
                           lpszMessage);  
  
CMessageStore mailStorage;  
  
if (mailStorage.OpenFile(lpszFileName, MIME_STORAGE_WRITE))  
{  
    LONG nLastMessage = 0;  
  
    // Get the last message number in the message store  
    mailStorage.GetMessageCount(&nLastMessage);
```

```
if (nLastMessage > 0)
{
    // Replace the last message in the message store
    if (!mailStorage.ReplaceMessage(nLastMessage, mailMessage))
    {
        // We were unable to replace the message
    }
}

// Close the message store
mailStorage.CloseFile();
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

See Also

[FindMessage](#), [GetMessage](#), [DeleteMessage](#), [StoreMessage](#)

CMessageStore::SetLastError Method

```
VOID SetLastError(  
    DWORD dwErrorCode  
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_MESSAGESTORE or MIME_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

See Also

[GetErrorString](#), [GetLastError](#), [ShowError](#)

CMessageStore::ShowError Method

```
INT ShowError(  
    LPCTSTR lpszAppTitle,  
    UINT uType,  
    DWORD dwErrorCode  
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A pointer to a string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetErrorString](#), [GetLastError](#), [SetLastError](#)

CMessageStore::StoreMessage Method

```
LONG StoreMessage(  
    CMailMessage& mailMessage  
);  
  
LONG StoreMessage(  
    HMESSAGE hMessage  
);
```

The **StoreMessage** method stores the specified message in the current message store.

Parameters

hStorage

Handle to the message store.

mailMessage

A **CMailMessage** object which references the message that will be stored.

hMessage

Handle to the message that will be stored.

Return Value

If the method succeeds, the return value is the message number for the message that was just stored. If the method fails, the return value is `MIME_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **StoreMessage** method will always append the specified message to the storage file. If you want to replace a message in the message store, you should use the **DeleteMessage** method to mark the original message for deletion and then use **StoreMessage** to write the updated message to the message store.

Example

```
CMailMessage mailMessage;  
  
// Compose a new message  
mailMessage.ComposeMessage(lpszSender,  
                           lpszRecipient,  
                           NULL,  
                           lpszSubject,  
                           lpszMessage,  
                           NULL,  
                           MIME_CHARSET_DEFAULT,  
                           MIME_ENCODING_DEFAULT);  
  
CMessageStore mailStorage;  
  
// Open the message storage file  
if (mailStorage.OpenFile(lpszFileName, MIME_STORAGE_WRITE))  
{  
    // Store a copy of the message in the message store  
    nMessageId = mailStorage.StoreMessage(mailMessage);  
  
    if (nMessageId == MIME_ERROR)
```

```
{  
    // We were unable to store the message  
}  
  
// Close the message store  
mailStorage.CloseFile();  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

See Also

[DeleteMessage](#), [FindMessage](#), [GetMessage](#)

Time Protocol Class Library

Query a time server for the current time and synchronize the local system clock with that value.

Reference

- [Class Methods](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

Class Name	CNetworkTime
File Name	CSTIMV10.DLL
Version	10.0.1468.2518
LibID	7E65DD9F-5799-42FA-A72D-EC0B714E3021
Import Library	CSTIMV10.LIB
Dependencies	None
Standards	RFC 868

Overview

The Time Protocol class provides an interface for synchronizing the local system's time and date with that of a server. The time values returned are in Coordinated Universal and be adjusted for the local host's timezone. The library enables developers to query a server for the current time and then update the system clock if desired.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Time Protocol Class Methods

Class	Description
CNetworkTime	Constructor which initializes the current instance of the class
~CNetworkTime	Destructor which releases resources allocated by the class
Method	Description
ConvertTime	Convert between network and system time values
DisableTrace	Disable logging of socket function calls to the trace log
EnableTrace	Enable logging of socket function calls to a file
GetErrorString	Return a description for the specified error code
GetLastError	Return the last error code
GetTime	Return the current network time from the server
IsInitialized	Determine if the class has been successfully initialized
SetLastError	Set the last error code
SetTime	Set the local system time with the network time

CNetworkTime::CNetworkTime Method

```
CNetworkTime();
```

The **CNetworkTime** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **TimeInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstimv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[~CNetworkTime](#), [IsInitialized](#)

CNetworkTime::~~CNetworkTime

`~CNetworkTime();`

The **CNetworkTime** destructor releases resources allocated by the current instance of the **CNetworkTime** object. It also uninitialized the library if there are no other concurrent uses of the class.

Remarks

When a **CNetworkTime** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstimv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CNetworkTime](#)

CNetworkTime::ConvertTime Method

```
BOOL ConvertTime(  
    DWORD dwNetworkTime,  
    LPLONG lpnUnixTime  
);  
  
BOOL ConvertTime(  
    DWORD dwNetworkTime,  
    CTime& LocalTime  
);  
  
BOOL ConvertTime(  
    DWORD dwNetworkTime,  
    LPSYSTEMTIME lpSystemTime,  
    BOOL bLocalTime  
);  
  
BOOL ConvertTime(  
    LPSYSTEMTIME lpSystemTime,  
    LPDWORD lpdwNetworkTime  
);
```

The **ConvertTime** method converts between a 32-bit network time value and a SYSTEMTIME structure.

Parameters

dwNetworkTime

The network time to be converted.

lpnUnixTime

A pointer to a long integer which will contain the time in UNIX format. The value is the number of seconds since 1 January 1970 UTC and is commonly used with the standard C library time functions.

localTime

A **CTime** object which will contain the local time when the method returns.

lpSystemTime

A pointer to a **SYSTEMTIME** structure which will be modified for the specified network time.

bLocalTime

A boolean flag that is used to specify if the network time should be adjusted for the local timezone.

Return Value

If the network time could be converted, the method returns a non-zero value. If the network time cannot be converted, or the pointer to the SYSTEMTIME structure is invalid, the method will return zero.

Remarks

The network time value is a 32-bit number, represented as the number of seconds since midnight, 1 January 1900 UTC. It can represent a date and time up to the year 2036.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstimv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetTime](#), [SetTime](#)

CNetworkTime::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstimv10.lib

See Also

[EnableTrace](#)

CNetworkTime::EnableTrace Method

```
BOOL EnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **EnableTrace** method enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace method logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstimv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableTrace](#)

CNetworkTime::GetErrorString Method

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);  
  
INT GetErrorString(  
    DWORD dwErrorCode,  
    CString& strDescription  
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstimv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLastError](#), [SetLastError](#), [ShowError](#)

CNetworkTime::GetLastError Method

```
DWORD GetLastError();  
  
DWORD GetLastError(  
    CString& strDescription  
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError(0)** when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or TIME_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cstimv10.lib

See Also

[GetErrorString](#), [SetLastError](#), [ShowError](#)

CNetworkTime::GetTime Method

```
BOOL GetTime(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    LPDWORD lpdwNetworkTime  
    UINT nTimeout  
);  
  
BOOL GetTime(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    LPSYSTEMTIME lpSystemTime  
    UINT nTimeout  
);  
  
BOOL GetTime(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    CTime& LocalTime  
    UINT nTimeout  
);
```

The **GetTime** method returns the network time from the specified host.

Parameters

lpszRemoteHost

A pointer to the name of the server. The host must be running a time server that complies with the specifications outlined in RFC 868.

nRemotePort

The port the time server is running on. A value of zero indicates that the default port number for the service should be used. By default, port 37 is used to connect to the time server.

lpdwNetworkTime

A pointer to an unsigned 32-bit integer which will contain the current network time when the method returns. Note that this value is not the same as UNIX time and cannot be used with the standard C time functions.

lpSystemTime

A pointer to a [SYSTEMTIME](#) structure that will contain the system time when the method returns. This structure can be used with many of the standard Windows API time-related functions.

localTime

A **CTime** object that will contain the local time when the method returns.

nTimeout

The number of seconds that the method will wait for a response from the server.

Return Value

If the method succeeds, it returns the number of seconds since midnight, 1 January 1900 UTC. If the method was unable to obtain the time from the specified host, it returns zero.

Remarks

The network time is a 32-bit number, represented as the number of seconds since midnight, 1 January 1900 UTC. It can represent a date and time up to the year 2036. It is important to note that the network time value is not the same as the UNIX time value that is used the standard C library time functions. To convert between network time and other time values, use the **ConvertTime** method.

In the United States, the National Institute of Standards and Technology (NIST) hosts a number of public servers which can be used to obtain the current time. The following table lists the current host names and addresses:

Server Name	IP Address	Location
time-a.nist.gov	129.6.15.28	Gaithersburg, Maryland
time-b.nist.gov	129.6.15.29	Gaithersburg, Maryland
time-nw.nist.gov	131.107.13.100	Redmond, Washington
time-a.timefreq.bldrdoc.gov	132.163.4.101	Boulder, Colorado
time-b.timefreq.bldrdoc.gov	132.163.4.102	Boulder, Colorado
time-c.timefreq.bldrdoc.gov	132.163.4.103	Boulder, Colorado

Time servers are also commonly maintained by Internet service providers and universities. If you are unable to obtain the time from a server, contact the system administrator to determine if they have the standard time service available on port 37.

Example

```
CNetworkTime netTime;
DWORD dwNetworkTime;

// Get the current time from a NIST time server
if (netTime.GetTime(_T("time-nw.nist.gov"), &dwNetworkTime))
{
    // Convert the network time value to a CTime object
    CTime localTime;
    netTime.ConvertTime(dwNetworkTime, localTime);

    // Format the time string and create a message to display
    // to the user, asking if they want to update the time
    CString strTime;
    strTime = localTime.Format(_T("%B %d, %Y %H:%M:%S"));

    CString strMessage;
    strMessage.Format(_T("Update system time to %s?"), (LPCTSTR)strTime);

    INT nResult;
    nResult = AfxMessageBox(strMessage, MB_YESNO|MB_ICONQUESTION);

    if (nResult == IDYES)
    {
        // Update the local system time with the network time
        if (netTime.SetTime(dwNetworkTime) == FALSE)
            netTime.ShowError();
    }
}
else
{
```

```
    netTime.ShowError();  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstimv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ConvertTime](#), [SetTime](#)

CNetworkTime::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstimv10.lib

See Also

[CNetworkTime](#)

CNetworkTime::SetLastError Method

```
VOID SetLastError(  
    DWORD dwErrorCode  
);
```

The **SetLastError** method sets the error code for the current thread. This method is typically used to clear the last error by passing a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or TIME_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cstimv10.lib

See Also

[GetErrorString](#), [GetLastError](#)

CNetworkTime::SetTime Method

```
BOOL SetTime(  
    DWORD dwNetworkTime  
);  
  
BOOL SetTime(  
    LPSYSTEMTIME lpSystemTime  
);
```

The **SetTime** method sets the local system clock to the specified date and time.

Parameters

dwNetworkTime

The date and time the system clock should be set to, represented as the number of seconds since midnight, 1 January 1900.

lpSystemTime

A pointer to a [SYSTEMTIME](#) structure that specifies the date and time that the local clock should be set to.

Return Value

If the method is able to update the local time, it returns a non-zero value. If the specified time is invalid, or the user does not have the access rights to change the system clock, the method returns zero.

Remarks

The network time value is an unsigned 32-bit integer which can represent a date and time up to the year 2036. It is important to note that this is not the same as the UNIX time value which is used by the standard C library.

You must have administrator privileges in order to set the system clock.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstimv10.lib`

See Also

[ConvertTime](#), [GetTime](#)

CNetworkTime::ShowError Method

```
INT ShowError(  
    LPCTSTR lpszAppTitle,  
    UINT uType,  
    DWORD dwErrorCode  
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A pointer to a string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cstimv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetErrorString](#), [GetLastError](#), [SetLastError](#)

Time Protocol Data Structures

- SYSTEMTIME

SYSTEMTIME Structure

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include windows.h.

Network News Transfer Protocol Class Library

Download and submit articles to a news server.

Reference

- [Class Methods](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

Class Name	CNntpClient
File Name	CSNWSV10.DLL
Version	10.0.1468.2518
LibID	F37CEE4C-BAFE-43B9-B224-80975DF553B2
Import Library	CSNWSV10.LIB
Dependencies	None
Standards	RFC 977, RFC 2980

Overview

The Network News Transfer Protocol (NNTP) is used with servers that provide news services. This is similar in functionality to bulletin boards or message boards, where topics are organized hierarchically into groups, called newsgroups. Users can browse and search for messages, called news articles, which have been posted by other users. On many servers, they can also post their own articles which can be read by others. The largest collection of public newsgroups available is called USENET, a world-wide distributed discussion system. In addition, there are a large number of smaller news servers. For example, Catalyst Development operates a news server which methods as a forum for technical questions and announcements.

The SocketTools library provides a comprehensive interface for accessing newsgroups, retrieving articles and posting new articles. In combination with the Mail Message library to process the news articles, SocketTools can be used to integrate newsgroup access with an existing email application, or you can implement your own full-featured newsgroup client.

This library supports secure connections using the standard SSL and TLS protocols.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-

bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Network News Transfer Protocol Class Methods

Class	Description
CNntpClient	Constructor which initializes the current instance of the class
~CNntpClient	Destructor which releases resources allocated by the class
Method	Description
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified client handle to another thread
Authenticate	Authenticate the specified user on the news server
Cancel	Cancel the current blocking operation
CloseArticle	Close the article being posted to the current newsgroup
Command	Send a command to the server
Connect	Connect to the specified server
CreateArticle	Create a new article in the current newsgroup
CreateSecurityCredentials	Create a new security credentials structure
DeleteSecurityCredentials	Delete a previously created security credentials structure
DetachHandle	Detach the handle for the current instance of this class
DisableEvents	Disable asynchronous event notification
DisableTrace	Disable logging of socket function calls to the trace log
Disconnect	Disconnect from the current server
EnableEvents	Enable asynchronous event notification
EnableTrace	Enable logging of socket function calls to a file
FreezeEvents	Suspend asynchronous event processing
GetArticle	Copy the specified article to a local buffer
GetArticleHeaders	Return the contents of the specified article header
GetArticleMessageId	Return the message identifier for the specified article
GetArticleRange	Return the first and last article number for the current group
GetCurrentArticle	Return the current article number for the selected group
GetCurrentDate	Return the current date and time
GetErrorString	Return a description for the specified error code
GetFirstArticle	Return the first available article in the currently selected newsgroup
GetFirstGroup	Return the first available newsgroup from the server
GetHandle	Return the client handle used by this instance of the class
GetGroupName	Return the name of the currently selected newsgroup

GetGroupTitle	Return a description of the currently selected newsgroup
GetLastError	Return the last error code
GetMessageIdArticle	Return the article number for the specified message identifier
GetNextArticle	Return the next available article from the currently selected newsgroup
GetNextGroup	Return the next available newsgroup from the server
GetResultCode	Return the result code from the previous command
GetResultString	Return the result string from the previous command
GetSecurityInformation	Return security information about the current client connection
GetStatus	Return the current client status
GetTimeout	Return the number of seconds until an operation times out
GetTransferStatus	Return data transfer statistics
IsBlocking	Determine if the client is blocked, waiting for information
IsConnected	Determine if the client is connected to the server
IsInitialized	Determine if the class has been successfully initialized
IsReadable	Determine if data can be read from the server
IsWritable	Determine if data can be written to the server
ListArticles	Return a list of articles in the currently selected newsgroup
ListGroups	Return a list of newsgroups maintained by the server
NntpEventProc	Callback method that processes events generated by the client
OpenArticle	Open the specified article in the currently selected newsgroup
OpenNextArticle	Open the next available article
OpenPreviousArticle	Open the previous article
PostArticle	Post a new article to the news server
Read	Read data returned by the news server
RegisterEvent	Register an event callback function
Reset	Reset the client
SelectGroup	Select the specified newsgroup to retrieve articles from
SetLastError	Set the last error code
SetTimeout	Set the number of seconds until an operation times out
ShowError	Display a message box with a description of the specified error
StoreArticle	Store the specified article to a file on the local system
Write	Write data to the news server

CNntpClient::CNntpClient Method

CNntpClient();

The **CNntpClient** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **NntpInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[~CNntpClient](#), [IsInitialized](#)

CNntpClient::~CNntpClient

`~CNntpClient();`

The **CNntpClient** destructor releases resources allocated by the current instance of the **CNntpClient** object. It also uninitialized the library if there are no other concurrent uses of the class.

Remarks

When a **CNntpClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csnwsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CNntpClient](#)

CNntpClient::AttachHandle Method

```
VOID AttachHandle(  
    HCLIENT hClient  
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle already has been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the *hClient* parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[AttachThread](#), [DetachHandle](#), [GetHandle](#)

CNntpClient::AttachThread Method

```
DWORD AttachThread(  
    DWORD dwThreadId  
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[AttachHandle](#), [Cancel](#), [Connect](#), [DetachHandle](#), [Disconnect](#), [GetHandle](#)

CNntpClient::Authenticate Method

```
INT Authenticate(  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword  
);
```

The **Authenticate** method is used to authenticate access to the news server. Not all news servers require authentication by the client.

Parameters

lpszUserName

Pointer to a string which specifies the user name required for authentication on the news server.

lpszPassword

Pointer to a string which specifies the password required for authentication on the news server.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should only be called if the server requires authentication. Two authentication methods, "original" and "simple" authentication, are recognized by the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [ListArticles](#), [ListGroupes](#)

CNntpClient::Cancel Method

```
INT Cancel();
```

The **Cancel** method cancels any outstanding blocking operation in the client, causing the blocking method to fail. The application may then retry the operation or terminate the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[IsBlocking](#)

CNntpClient::CloseArticle Method

```
INT CloseArticle();
```

The **CloseArticle** method closes the current article that has been opened or created.

Parameters

None.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If an article is being created, this method actually submits the article to the server. Note that the client application is responsible for generating the message headers as well as the body of the message. News articles conform to the same general characteristics of an email message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[CreateArticle](#), [OpenArticle](#), [Write](#)

CNntpClient::Command Method

```
INT Command(  
    LPCTSTR lpszCommand,  
    LPCTSTR lpszParameter,  
    BOOL bMultiLine  
);
```

The **Command** method sends a command to the server, and returns the result code back to the caller. This method is typically used for site-specific commands not directly supported by the class.

Parameters

lpszCommand

The command which will be executed by the server.

lpszParameter

An optional command parameter. If the command requires more than one parameter, then they should be combined into a single string, with a space separating each parameter. If the command does not accept any parameters, this value may be NULL.

bMultiLine

An optional boolean argument used to specify if multiple lines of data will be returned by the server as the result of the command. Unlike a single line response, which consists of a result code and result string, a multi-line response consists of one or more lines of text, terminated by a special end-of-data marker. If this argument is omitted, *bMultiLine* is FALSE.

Return Value

If the method succeeds, the return value is the result code returned by the server. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

A list of valid commands can be found in the technical specification for the protocol. Many servers will list supported commands when the HELP command is used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetResultCode](#), [GetResultString](#)

CNntpClient::Connect Method

```
BOOL Connect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **Connect** method establishes a connection with the specified server and defines security related options to be used.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used. For standard connections, the default port number is NN. For secure connections, the default port number is NN.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
NNTP_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
NNTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
NNTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
NNTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure

	connections using either the SSL or TLS protocol.
NNTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
NNTP_OPTION_FREETHREAD	This option specifies that this instance of the class may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the class instance is synchronized across multiple threads.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur. If this argument is NULL, then the client session will be blocking and no network events will be sent to the client.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the *hEventWnd* argument is NULL, this argument should be specified as WM_NULL.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The use of Windows event notification messages has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

To prevent this method from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **Connect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

If you specify an event notification window, then the client session will be asynchronous. When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
NNTP_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
NNTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.

NNTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
NNTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
NNTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
NNTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
NNTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
NNTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

To cancel asynchronous notification and return the client to a blocking mode, use the **DisableEvents** methods.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the class instance is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the NNTP_OPTION_FREETHREAD option enables any thread to call any method in that instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same instance.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateSecurityCredentials](#), [Disconnect](#), [GetSecurityInformation](#)

CNntpClient::CreateArticle Method

```
INT CreateArticle();
```

The **CreateArticle** method creates a new article in the current newsgroup.

Parameters

None.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method sends the POST command to the news server. Not all servers permit clients to post articles. The client application is responsible for generating the message headers as well as the body of the message. News articles conform to the same general characteristics of an email message.

The **CloseArticle** method must be called once the contents of the article has been written to the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[CloseArticle](#), [GetCurrentDate](#), [ListArticles](#), [OpenArticle](#), [OpenNextArticle](#), [OpenPreviousArticle](#), [Write](#)

CNntpClient::CreateSecurityCredentials Method

```
BOOL CreateSecurityCredentials(  
    DWORD dwProtocol,  
    DWORD dwOptions,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName  
);  
  
BOOL CreateSecurityCredentials(  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName  
);  
  
BOOL CreateSecurityCredentials(  
    LPCTSTR lpszCertName  
);
```

The **CreateSecurityCredentials** method establishes the security credentials for the client session.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols.

	This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpzUserName

A pointer to a string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A pointer to a string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A pointer to a string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Example

```
pClient->CreateSecurityCredentials(  
    SECURITY_PROTOCOL_DEFAULT,  
    0,  
    NULL,  
    NULL,  
    lpszCertStore,  
    lpszCertName);
```

```
bConnected = pClient->Connect(lpszHostName,  
                               NNTP_PORT_SECURE,  
                               NNTP_TIMEOUT,  
                               NNTP_OPTION_SECURE);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [DeleteSecurityCredentials](#), [GetSecurityInformation](#), [SECURITYCREDENTIALS](#)

CNntpClient::DeleteSecurityCredentials Method

```
VOID DeleteSecurityCredentials();
```

The **DeleteSecurityCredentials** method releases the security credentials for the current session.

Parameters

None.

Return Value

None.

Remarks

This method can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated. The security credentials are released when the class destructor is called, so it is normally not required that the application explicitly call this method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateSecurityCredentials](#)

CNntpClient::DetachHandle Method

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csnwsv10.lib`

See Also

[AttachHandle](#), [GetHandle](#)

CNntpClient::DisableEvents Method

```
INT DisableEvents();
```

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the client has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[EnableEvents](#), [RegisterEvent](#)

CNntpClient::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[EnableTrace](#)

CNntpClient::Disconnect Method

VOID Disconnect();

The **Disconnect** method terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[Connect](#), [IsConnected](#)

CNntpClient::EnableEvents Method

```
INT EnableEvents(  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Constant	Description
NNTP_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
NNTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
NNTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
NNTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking

	operation. This event is only generated if the client is in asynchronous mode.
NNTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
NNTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
NNTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
NNTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

It is not required that the client be placed in asynchronous mode in order to receive command and progress event notifications. To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnews10.lib

See Also

[DisableEvents](#), [RegisterEvent](#)

CNntpClient::EnableTrace Method

```
BOOL EnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **EnableTrace** method enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace method logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableTrace](#)

CNntpClient::FreezeEvents Method

```
INT FreezeEvents(  
    BOOL bFreeze  
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the client.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableEvents](#), [EnableEvents](#), [RegisterEvent](#)

CNntpClient::GetArticle Method

```
INT GetArticle(  
    LONG nArticleId,  
    LPBYTE lpBuffer,  
    LPDWORD lpdwLength  
);
```

```
INT GetArticle(  
    LONG nArticleId,  
    HGLOBAL* lpBuffer,  
    LPDWORD lpdwLength  
);
```

```
INT GetArticle(  
    LONG nArticleId,  
    CString& strBuffer  
);
```

```
INT GetArticle(  
    LPCTSTR lpszMessageId,  
    LPBYTE lpBuffer,  
    LPDWORD lpdwLength  
);
```

```
INT GetArticle(  
    LPCTSTR lpszMessageId,  
    HGLOBAL* lpBuffer,  
    LPDWORD lpdwLength  
);
```

```
INT GetArticle(  
    LPCTSTR lpszMessageId,  
    CString& strBuffer  
);
```

The **GetArticle** method retrieves the specified article and copies the contents to a local buffer.

Parameters

nArticleId

An integer value which specifies the number of the article to retrieve from the server. This value must be greater than zero. Overloaded versions of this method also support article IDs that are 64-bit integers.

lpszMessageId

A pointer to a string which specifies the message ID of the article to retrieve from the server. This parameter cannot be NULL or specify an empty string.

lpBuffer

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the method returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpBuffer* parameter. If the *lpBuffer* parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual length of the file that was

downloaded.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetArticle** method is used to retrieve an article from the server and copy it into a local buffer. The method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the article. In this case, the *lpBuffer* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpBuffer* parameter point to a global memory handle which will contain the article data when the method returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. See the example code below.

This method will cause the current thread to block until the complete article has been retrieved, a timeout occurs or the operation is canceled. During the transfer, the NNTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a transfer while it is in progress, use the **GetTransferStatus** method.

Example

```
HGLOBAL hgb1Buffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Return the article into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgb1Buffer parameter
nResult = pClient->GetArticle(nArticleId, &hgb1Buffer, &cbBuffer);

if (nResult != NNTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // article text
    lpBuffer = (LPBYTE)GlobalLock(hgb1Buffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgb1Buffer);
    GlobalFree(hgb1Buffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateArticle](#), [EnableEvents](#), [GetArticleHeaders](#), [GetTransferStatus](#), [RegisterEvent](#)

CNntpClient::GetArticleHeaders Method

```
INT GetArticleHeaders(  
    LONG nArticleId,  
    LPBYTE lpHeaders,  
    LPDWORD lpdwLength  
);
```

```
INT GetArticleHeaders(  
    LONG nArticleId,  
    HGLOBAL* lpHeaders,  
    LPDWORD lpdwLength  
);
```

```
INT GetArticleHeaders(  
    LONG nArticleId,  
    CString& strHeaders  
);
```

The **GetArticleHeaders** method retrieves the headers for the specified article from the server.

Parameters

nArticleId

Number of article to retrieve from the server. This value must be greater than zero. Overloaded versions of this method also support article IDs that are 64-bit integers.

lpHeaders

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the method returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpHeaders* parameter. If the *lpHeaders* parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual length of the message that was downloaded.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetArticleHeaders** method is used to retrieve an article header block from the server and copy it into a local buffer. The method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the *lpHeaders* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpHeaders* parameter point to a global memory handle which will contain the message headers when the method returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur.

This method will cause the current thread to block until the transfer completes, a timeout occurs

or the transfer is canceled. During the transfer, the NNTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[GetArticle](#), [GetArticleRange](#), [ListArticles](#), [PostArticle](#)

CNntpClient::GetArticleMessageId Method

```
INT GetArticleMessageId(  
    LONG nArticleId,  
    LPTSTR lpszMessageId,  
    INT cbMessageId  
);  
  
INT GetArticleMessageId(  
    LONG nArticleId,  
    CString& strMessageId  
);
```

The **GetArticleMessageId** method returns the message identifier for the specified article in the current newsgroup.

Parameters

nArticleId

Article number to retrieve the message identifier for. The value may be zero, in which case the current article number is used.

lpszMessageId

Pointer to a string buffer which will contain the message identifier for the specified article.

cbMessageId

Maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the message identifier string. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The message identifier is a string which can uniquely identify the message on the news server. This value may be used to retrieve the contents of the article.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetMessageIdArticle](#), [ListArticles](#)

CNntpClient::GetArticleRange Method

```
BOOL GetArticleRange(  
    LONG * lpnFirstArticle,  
    LONG * lpnLastArticle,  
    LONG * lpnArticleCount  
);  
  
BOOL GetArticleRange(  
    ULONGLONG * lpnFirstArticle,  
    ULONGLONG * lpnLastArticle  
    ULONGLONG * lpnArticleCount  
);
```

The **GetArticleRange** method returns the first and last article numbers for the currently selected newsgroup.

Parameters

lpnFirstArticle

Pointer to an integer that will contain the first article number in the currently selected newsgroup. If this parameter is NULL, it will be ignored.

lpnLastArticle

Pointer to an integer that will contain the last article number in the currently selected newsgroup. If this parameter is NULL, it will be ignored.

lpnArticleCount

Pointer to an integer that will contain the total number of articles in currently selected newsgroup. If this parameter is NULL, it will be ignored.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

It is possible that there will be gaps in the articles within the range of the first and last articles in the newsgroup. This may be due to a message being canceled or expired. If the server can potentially return very large ID values, it is recommended your application use ULONGLONGs (64-bit unsigned integers) instead of LONGs (32-bit signed integers) to store article numbers.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[OpenArticle](#), [ListArticles](#)

CNntpClient::GetCurrentArticle Method

```
BOOL GetCurrentArticle(  
    LONG * lpnArticleId  
);  
  
BOOL GetCurrentArticle(  
    ULONGLONG * lpnArticleId  
);
```

The **GetCurrentArticle** method returns the current article number for the currently selected newsgroup.

Parameters

lpnArticleId

Pointer to an integer variable that will contain the current article number when the method returns.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

If the server can potentially return very large ID values, it is recommended your application use ULONGLONGs (64-bit unsigned integers) instead of LONGs (32-bit signed integers) to store article numbers.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[OpenArticle](#), [GetArticleRange](#), [GetFirstArticle](#), [GetNextArticle](#), [ListArticles](#)

CNntpClient::GetCurrentDate Method

```
INT GetCurrentDate(  
    LPTSTR lpszDate,  
    INT nMaxLength  
);  
  
INT GetCurrentDate(  
    CString& strDate  
);
```

The **GetCurrentDate** method copies the current date and time to the specified buffer in a format that is commonly used in news articles. This date format should be used in all date-related fields in the message header.

Parameters

lpszDate

Pointer to a string buffer that will contain the current date and time when the method returns.

nMaxLength

The maximum number of characters that can be copied into the string buffer.

Return Values

If the method succeeds, the return value is the number of characters copied into the buffer, not including the null-terminator. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The date value that is returned is adjusted for the local timezone.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateArticle](#)

CNntpClient::GetErrorString Method

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);  
  
INT GetErrorString(  
    DWORD dwErrorCode,  
    CString& strDescription  
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLastError](#), [SetLastError](#), [ShowError](#)

CNntpClient::GetFirstArticle Method

```
BOOL GetFirstArticle(  
    LPNEWSARTICLE lpArticle  
);  
  
BOOL GetFirstArticle(  
    LPNEWSARTICLEEX lpArticle  
);
```

The **GetFirstArticle** method returns information about the first article in the currently selected newsgroup.

Parameters

lpArticle

A pointer to a [NEWSARTICLE](#) or [NEWSARTICLEEX](#) structure which will contain information about the first article in the currently selected directory. If the server can potentially return very large article IDs, it is recommended that you use the **NEWSARTICLEEX** structure.

Return Value

If the method succeeds, the return value is non-zero. If there are no articles in the current newsgroup, or the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetFirstArticle** method returns information about the first article in the currently selected newsgroup. This method is used in conjunction with the **GetNextArticle** method to enumerate all of the articles in the newsgroup. Typically this is used to provide the user with a list of articles to access.

While the articles in the newsgroup are being listed, the client cannot retrieve the contents of a specific article. For example, the **GetArticle** method cannot be called while inside a loop calling **GetNextArticle**. The client should store those articles which it wants to retrieve in an array, and then once all of the articles have been listed, it can begin calling **NntpGetArticle** for each article number to retrieve the article text.

The date and time that the article was posted is returned in the *stPosted* member of the **NEWSARTICLE** structure. This value is returned in Universal Coordinated Time (UTC) and can be converted to local time using the **SystemTimeToTzSpecificLocalTime** function.

Example

```
// List all articles in the current group  
if (pClient->ListArticles() == NNTP_ERROR)  
    pClient->ShowError();  
else  
{  
    NEWSARTICLE newsArticle;  
    BOOL bResult;  
  
    // Get each article in the current newsgroup, printing the article  
    // number and the subject of the article  
    bResult = pClient->GetFirstArticle(&newsArticle);  
    while (bResult)  
    {
```

```
        _tprintf(_T("%ld %s\n"), newsArticle.nArticleId, newsArticle.szSubject);  
        bResult = pClient->GetNextArticle(&newsArticle);  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetArticle](#), [GetFirstGroup](#), [GetNextArticle](#), [GetNextGroup](#), [ListArticles](#), [ListGroup](#), [SelectGroup](#), [NEWSARTICLE](#), [NEWSGROUP](#)

CNntpClient::GetFirstGroup Method

```
BOOL GetFirstGroup(  
    LPNEWSGROUP LpGroup  
);  
  
BOOL GetFirstGroup(  
    LPNEWSGROUPEX LpGroup  
);
```

The **GetFirstGroup** method returns information about the first available newsgroup.

Parameters

lpGroup

A pointer to a [NEWSGROUP](#) or [NEWSGROUPEX](#) structure which will contain information about the first available newsgroup. If the server can potentially return very large article IDs, it is recommended that you use the [NEWSGROUPEX](#) structure.

Return Value

If the method succeeds, the return value is non-zero. If there are no newsgroups available, or the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetFirstGroup** method returns information about the first newsgroup on the server. This method is used in conjunction with the **GetNextGroup** method to enumerate all of the available newsgroups. Typically this is used to provide the user with a list of newsgroups to select.

While the the newsgroups are being listed, the client cannot select a newsgroup or retrieve the contents of a specific article. The client should store those newsgroups which it wants to retrieve articles from, and then once all of the newsgroups have been listed, it can then select each newsgroup and retrieve the available articles from that group.

Note that if no newsgroups are returned by the server, it may indicate that it requires the client to authenticate itself prior to requesting a list of groups or articles.

Example

```
// List all available newsgroups  
if (pClient->ListGroups() == NNTP_ERROR)  
    pClient->ShowError();  
else  
{  
    NEWSGROUP newsGroup;  
    BOOL bResult;  
  
    // Get each newsgroup, printing the article range and  
    // the name of the group  
    bResult = pClient->GetFirstGroup(&newsGroup);  
    while (bResult)  
    {  
        printf("%ld %ld %s\n", newsGroup.nFirstArticle,  
            newsGroup.nLastArticle,  
            newsGroup.szName);  
        bResult = pClient->GetNextGroup(&newsGroup);  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFirstArticle](#), [GetNextArticle](#), [GetNextGroup](#), [ListArticles](#), [ListGroup](#), [SelectGroup](#), [NEWSARTICLE](#), [NEWSGROUP](#)

CNntpClient::GetGroupName Method

```
INT GetGroupName(  
    LPTSTR lpszGroupName,  
    INT nMaxLength  
);
```

```
INT GetGroupName(  
    CString& strGroupName  
);
```

The **GetGroupName** method returns the name of the currently selected newsgroup.

Parameters

lpszGroupName

Pointer to a string buffer that will contain the name of the currently selected newsgroup.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the newsgroup name. If no newsgroup has been selected, the method will return a value of zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetGroupTitle](#), [ListGroup](#), [SelectGroup](#)

CNntpClient::GetGroupTitle Method

```
INT GetGroupTitle(  
    LPTSTR lpszGroupTitle,  
    INT nMaxLength  
);  
  
INT GetGroupTitle(  
    CString& strGroupTitle  
);
```

The **GetGroupTitle** method returns a description of the currently selected newsgroup.

Parameters

lpszGroupTitle

Pointer to a string buffer that will contain a description of the currently selected newsgroup.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the description. If no newsgroup has been selected, the method will return a value of zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The news server must support the XGTITLE command so that the group description can be obtained when the newsgroup is selected. If this command is not recognized, then no description will be returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csnews10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetGroupName](#), [ListGroup](#), [SelectGroup](#)

CNntpClient::GetHandle Method

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csnwsv10.lib`

See Also

[AttachHandle](#), [DetachHandle](#), [IsInitialized](#)

CNntpClient::GetLastError Method

```
DWORD GetLastError();  
  
DWORD GetLastError(  
    CString& strDescription  
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError(0)** when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or NNTP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[GetErrorString](#), [SetLastError](#), [ShowError](#)

CNntpClient::GetMessageIdArticle Method

```
BOOL GetMessageIdArticle(  
    LPCTSTR LpszMessageId,  
    LONG * LpnMessageId  
);  
  
BOOL GetMessageIdArticle(  
    LPCTSTR LpszMessageId,  
    ULONGLONG * LpnMessageId  
);
```

The **GetMessageIdArticle** method returns the article number associated with the specified message identifier in the current newsgroup.

Parameters

lpszMessageId

A pointer to the message identifier string.

lpnArticleId

Pointer to an integer variable that will contain the article number associated with the message ID.

Return Value

If the method succeeds, the return value is the article number. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the server can potentially return very large ID values, it is recommended your application use ULONGLONGs (64-bit unsigned integers) instead of LONGs (32-bit signed integers) to store article numbers.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetArticleMessageId](#), [ListArticles](#)

CNntpClient::GetNextArticle Method

```
BOOL GetNextArticle(  
    LPNEWSARTICLE lpArticle  
);  
  
BOOL GetNextArticle(  
    LPNEWSARTICLEEX lpArticle  
);
```

The **GetNextArticle** method returns information about the next article in the currently selected newsgroup.

Parameters

lpArticle

A pointer to a [NEWSARTICLE](#) or [NEWSARTICLEEX](#) structure which will contain information about the next available article in the currently selected directory. If the server can potentially return very large article IDs, it is recommended that you use the **NEWSARTICLEEX** structure.

Return Value

If the method succeeds, the return value is non-zero. If there are no more articles in the current newsgroup, or the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetNextArticle** method returns information about the next available article in the currently selected newsgroup. This method is used in conjunction with the **GetFirstArticle** method to enumerate all of the articles in the newsgroup. Typically this is used to provide the user with a list of articles to access.

While the articles in the newsgroup are being listed, the client cannot retrieve the contents of a specific article. For example, the **GetArticle** method cannot be called while inside a loop calling **GetNextArticle**. The client should store those articles which it wants to retrieve in an array, and then once all of the articles have been listed, it can begin calling **NntpGetArticle** for each article number to retrieve the article text.

Example

```
// List all articles in the current group  
if (pClient->ListArticles() == NNTP_ERROR)  
    pClient->ShowError();  
else  
{  
    NEWSARTICLE newsArticle;  
    BOOL bResult;  
  
    // Get each article in the current newsgroup, printing the article  
    // number and the subject of the article  
    bResult = pClient->GetFirstArticle(&newsArticle);  
    while (bResult)  
    {  
        _tprintf(_T("%ld %s\n"), newsArticle.nArticleId, newsArticle.szSubject);  
        bResult = pClient->GetNextArticle(&newsArticle);  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetArticle](#), [GetFirstArticle](#), [GetFirstGroup](#), [GetNextGroup](#), [ListArticles](#), [ListGroup](#), [SelectGroup](#), [NEWSARTICLE](#), [NEWSGROUP](#)

CNntpClient::GetNextGroup Method

```
BOOL GetNextGroup(  
    LPNEWSGROUP LpGroup  
);  
  
BOOL GetNextGroup(  
    LPNEWSGROUPEX LpGroup  
);
```

The **GetNextGroup** method returns information about the next available newsgroup.

Parameters

lpGroup

A pointer to a [NEWSGROUP](#) or [NEWSGROUPEX](#) structure which will contain information about the next available newsgroup. If the server can potentially return very large article IDs, it is recommended that you use the **NEWSGROUPEX** structure.

Return Value

If the method succeeds, the return value is non-zero. If there are no more newsgroups available, or the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetNextGroup** method returns information about the next newsgroup on the server. This method is used in conjunction with the **GetFirstGroup** method to enumerate all of the available newsgroups. Typically this is used to provide the user with a list of newsgroups to select.

While the the newsgroups are being listed, the client cannot select a newsgroup or retrieve the contents of a specific article. The client should store those newsgroups which it wants to retrieve articles from, and then once all of the newsgroups have been listed, it can then select each newsgroup and retrieve the available articles from that group.

Example

```
// List all available newsgroups  
if (pClient->ListGroups() == NNTP_ERROR)  
    pClient->ShowError();  
else  
{  
    NEWSGROUP newsGroup;  
    BOOL bResult;  
  
    // Get each newsgroup, printing the article range and  
    // the name of the group  
    bResult = pClient->GetFirstGroup(&newsGroup);  
    while (bResult)  
    {  
        printf("%ld %ld %s\n", newsGroup.nFirstArticle,  
            newsGroup.nLastArticle,  
            newsGroup.szName);  
        bResult = pClient->GetNextGroup(&newsGroup);  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFirstArticle](#), [GetFirstGroup](#), [GetNextArticle](#), [ListArticles](#), [ListGroup](#), [SelectGroup](#), [NEWSARTICLE](#), [NEWSGROUP](#)

CNntpClient::GetResultCode Method

```
INT GetResultCode();
```

The **GetResultCode** method reads the result code returned by the server in response to a command. The result code is a three-digit numeric code, and indicates if the operation succeeded, failed or requires additional action by the client.

Parameters

None.

Return Value

If the method succeeds, the return value is the result code. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

Result codes are three-digit numeric values returned by the server. They may be broken down into the following ranges:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[Command](#), [GetResultString](#)

CNntpClient::GetResultString Method

```
INT GetResultString(  
    LPTSTR lpszResult,  
    INT cbResult  
);
```

```
INT GetResultString(  
    CString& strResult  
);
```

The **GetResultString** method returns the last message sent by the server along with the result code.

Parameters

lpszResult

A pointer to the buffer that will contain the result string returned by the server. An alternate form of the method accepts a **CString** argument which will contain the result string returned by the server.

cbResult

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetResultString** method is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **GetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnews10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Command](#), [GetResultCode](#)

CNntpClient::GetSecurityInformation Method

```
BOOL GetSecurityInformation(  
    LPSECURITYINFO LpSecurityInfo  
);
```

The **GetSecurityInformation** method returns security protocol, encryption and certificate information about the current client connection.

Parameters

LpSecurityInfo

A pointer to a [SECURITYINFO](#) structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;  
securityInfo.dwSize = sizeof(SECURITYINFO);  
  
if (pClient->GetSecurityInformation(&securityInfo))  
{  
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)  
    {  
        MessageBox(NULL, _T("The connection is not secure"),  
                    _T("Connection"), MB_OK);  
    }  
    else  
    {  
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)  
        {  
            MessageBox(NULL, _T("The connection is secure"),  
                        _T("Connection"), MB_OK);  
        }  
        else  
        {  
            MessageBox(NULL, _T("The server certificate not valid"),  
                        _T("Connection"), MB_OK);  
        }  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [CreateSecurityCredentials](#), [SECURITYINFO](#)

CNntpClient::GetStatus Method

INT GetStatus();

The **GetStatus** method the current status of the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the client status code. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetStatus** method returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Constant	Description
0	NNTP_STATUS_UNUSED	No connection has been established.
1	NNTP_STATUS_IDLE	The client is current idle and not sending or receiving data.
2	NNTP_STATUS_CONNECT	The client is establishing a connection with the server.
3	NNTP_STATUS_READ	The client is reading data from the server.
4	NNTP_STATUS_WRITE	The client is writing data to the server.
5	NNTP_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session. To obtain status information about a file transfer, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[IsBlocking](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#)

CNntpClient::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[Connect](#), [IsReadable](#), [IsWritable](#), [Read](#), [SetTimeout](#), [Write](#)

CNntpClient::GetTransferStatus Method

```
INT GetTransferStatus(  
    LPNNTPTRANSFERSTATUS LpStatus  
);  
  
INT GetTransferStatus(  
    LPNNTPTRANSFERSTATUSEX LpStatus  
);
```

The **GetTransferStatus** method returns information about the current data transfer in progress.

Parameters

LpStatus

A pointer to an [NNTPTRANSFERSTATUS](#) or [NNTPTRANSFERSTATUSEX](#) structure which contains information about the status of the current data transfer. If the server can potentially return very large article IDs, it is recommended that you use the **NNTPTRANSFERSTATUSEX** structure.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `NNTP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **GetTransferStatus** method returns information about the current data transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If no article is currently being retrieved or submitted to the server, this function will return the status of the last successful data transfer made by the client.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csnwsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableEvents](#), [GetStatus](#), [RegisterEvent](#)

CNntpClient::IsBlocking Method

BOOL IsBlocking();

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking method may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Cancel](#), [GetStatus](#), [IsConnected](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#), [Read](#), [Write](#)

CNntpClient::IsConnected Method

```
BOOL IsConnected();
```

The **IsConnected** method is used to determine if the client is currently connected to a server.

Parameters

None.

Return Value

If the client is connected to a server, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#)

CNntpClient::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[CNntpClient](#), [IsBlocking](#), [IsConnected](#)

CNntpClient::IsReadable Method

```
BOOL IsReadable(  
    INT nTimeout,  
    LPDWORD lpdwAvail  
);
```

The **IsReadable** method is used to determine if data is available to be read from the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the method returns a non-zero value. If the client cannot read any data, the method returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsConnected](#), [IsInitialized](#), [IsWritable](#), [Write](#)

CNntpClient::IsWritable Method

```
BOOL IsWritable(  
    INT nTimeout  
);
```

The **IsWritable** method is used to determine if data can be written to the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the method returns a non-zero value. If the client cannot write any data, the method returns zero.

Remarks

Although this method can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsConnected](#), [IsInitialized](#), [IsReadable](#), [Write](#)

CNntpClient::ListArticles Method

```
INT ListArticles(  
    LONG nFirstArticle,  
    LONG nLastArticle  
);  
  
INT ListArticles(  
    ULONGLONG nFirstArticle,  
    ULONGLONG nLastArticle  
);
```

The **ListArticles** method returns a list of articles in the currently selected newsgroup, within the specified article range.

Parameters

nFirstArticle

The first newsgroup article to be returned in the list. If this value is -1, the list will begin with the first available article in the newsgroup.

nLastArticle

The last newsgroup article to be returned in the list. If the value is -1, the list will end with the last available article in the newsgroup.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

It is possible that there will be gaps in the articles within the range of the first and last articles in the newsgroup. This may be due to a message being canceled or expired. Use the **GetFirstArticle** and **GetNextArticle** methods to read the list of articles returned by the server. If the server can potentially return very large ID values, it is recommended your application use ULONGLONGs (64-bit unsigned integers) instead of LONGs (32-bit signed integers) to store article numbers.

Example

```
// List all articles in the current group  
if (pClient->ListArticles() == NNTP_ERROR)  
    pClient->ShowError();  
else  
{  
    NEWSARTICLE newsArticle;  
    BOOL bResult;  
  
    // Get each article in the current newsgroup, printing the article  
    // number and the subject of the article  
    bResult = pClient->GetFirstArticle(&newsArticle);  
    while (bResult)  
    {  
        printf("%ld %s %s\n", newsArticle.nArticleId, newsArticle.szSubject);  
        bResult = pClient->GetNextArticle(&newsArticle);  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetArticleRange](#), [GetCurrentArticle](#), [GetFirstArticle](#), [GetNextArticle](#), [ListGroup](#)s

CNntpClient::ListGroup Method

```
INT ListGroups(  
    LPCTSTR lpszLastUpdated,  
    BOOL bLocalTime  
);
```

The **ListGroups** method instructs the server to begin sending a list of newsgroups that were created since the specified date.

Parameters

lpszLastUpdated

Pointer to a string which specifies the date and time that the list of newsgroups were last retrieved from the server. This parameter may be NULL or an empty string, in which case all available newsgroups will be listed by the server.

bLocalTime

A boolean value which indicates if the time specified in the *lpszLastUpdated* parameter is for the current timezone. If the value is non-zero, the time is assumed to be in the local timezone. If the value is zero, the time is assumed to be in Coordinated Universal Time (UTC).

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **ListGroups** method is used in conjunction with the **GetFirstGroup** and **GetNextGroup** methods to enumerate all of the newsgroups that were added to the server since a specific date and time. Typically this is used to provide the user with a list of updated newsgroups to select. To list all of the newsgroups available on the server, omit the arguments to this method.

While the newsgroups are being listed, the client cannot select a newsgroup or retrieve the contents of a specific article. The client should store those newsgroups which it wants to retrieve articles from, and then once all of the newsgroups have been listed, it can then select each newsgroup and retrieve the available articles from that group.

Example

```
LPCTSTR lpszUpdated = _T("1/1/2004 12:00 AM");  
  
// List all newsgroups that were added after a specific date  
if (pClient->ListGroup(lpszUpdated, TRUE) == NNTP_ERROR)  
    pClient->ShowError();  
else  
{  
    NEWSGROUP newsGroup;  
    BOOL bResult;  
  
    // Get each newsgroup, printing the article range and  
    // the name of the group  
    bResult = pClient->GetFirstGroup(&newsGroup);  
    while (bResult)  
    {  
        printf("%ld %ld %s\n", newsGroup.nFirstArticle,  
              newsGroup.nLastArticle,
```

```
        newsGroup.szName);  
    bResult = pClient->GetNextGroup(&newsGroup);  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetArticleRange](#), [GetFirstGroup](#), [GetNextGroup](#), [ListGroup](#), [SelectGroup](#)

CNntpClient::OpenArticle Method

```
INT OpenArticle(  
    LONG nArticleId  
);  
  
INT OpenArticle(  
    LPCTSTR lpszMessageId  
);
```

The **OpenArticle** method opens the specified article in the currently selected newsgroup.

Parameters

nArticleId

Number that specifies which article in the current newsgroup to retrieve. This value may be zero, which specifies that the current article should be returned.

lpszMessageId

Pointer to a string which contains the message identifier for the article in the current newsgroup.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[CloseArticle](#), [GetArticle](#), [GetArticleRange](#), [GetCurrentArticle](#), [ListArticles](#), [OpenNextArticle](#), [OpenPreviousArticle](#), [Read](#)

CNntpClient::OpenNextArticle Method

```
INT OpenNextArticle();
```

The **OpenNextArticle** method opens the next available article in the current newsgroup.

Parameters

None.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[CloseArticle](#), [GetArticle](#), [GetArticleRange](#), [GetCurrentArticle](#), [ListArticles](#), [OpenPreviousArticle](#), [Read](#)

CNntpClient::OpenPreviousArticle Method

```
INT OpenPreviousArticle();
```

The **OpenPreviousArticle** method opens the previous article in the current newsgroup.

Parameters

None.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[CloseArticle](#), [GetArticle](#), [GetArticleRange](#), [GetCurrentArticle](#), [ListArticles](#), [OpenNextArticle](#), [Read](#)

CNntpClient::PostArticle Method

```
INT PostArticle(  
    LPCTSTR lpBuffer,  
    DWORD dwLength  
);
```

The **PostArticle** method post the contents of the specified buffer to the server as a new article in the current newsgroup.

Parameters

lpBuffer

A pointer to a character buffer which contains the article to be posted to the currently selected newsgroup.

dwLength

Specifies the length of the string which contains the article. If this parameter is -1, the actual length of the string is calculated by searching the buffer for a terminating null byte.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **PostArticle** method is used to post the contents of the specified buffer to the server as a new article in the current newsgroup. Not all newsgroups permit new articles to be posted, and some newsgroups may require that you email the article to a moderator for approval instead of posting directly to the group. It may be required that the client authenticate itself using the **Authenticate** method prior to posting the article.

A news article is similar to an email message in that it contains one or more header fields, followed by an empty line, followed by the body of the article. Each line of text should be terminated by a carriage return/linefeed sequence of characters. The Mail Message library can be used to compose a message if needed. Note that the article header must contain a header field named "Newsgroups" with a value that specifies the newsgroup or newsgroups the article is being posted to. If this header field is missing, the news server will reject the article.

This method will cause the current thread to block until the transfer has completed, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a transfer while it is in progress, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

GetTransferStatus, IsBlocking, IsWritable, Read, Reset, Write

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CNntpClient::Read Method

```
INT Read(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Read(  
    CString& strBuffer,  
    INT cbBuffer  
);
```

The **Read** method reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be copied. An alternate form of this method allows a **CString** variable to be passed and data read from the socket will be returned in that string.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When **Read** is called and the client is in non-blocking mode, it is possible that the method will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[EnableEvents](#), [IsBlocking](#), [IsReadable](#), [IsWritable](#), [RegisterEvent](#), [Write](#)

CNntpClient::RegisterEvent Method

```
INT RegisterEvent(  
    UINT nEventId,  
    NNTPEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Constant	Description
NNTP_EVENT_CONNECT	The connection to the server has completed.
NNTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
NNTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
NNTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
NNTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
NNTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
NNTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
NNTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **NntpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **NntpEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This method is typically used to register an event handler that is invoked while a news article is being uploaded or downloaded. The NNTP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one NNTP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csnwsv10.lib

See Also

[DisableEvents](#), [EnableEvents](#), [FreezeEvents](#), [NntpEventProc](#)

CNntpClient::Reset Method

```
INT Reset();
```

The **Reset** method resets the client state and resynchronizes with the server. This method is typically called after an unexpected error has occurred, or an operation has been canceled.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The client cannot be reset while it is in a blocked state.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

See Also

[Cancel](#), [IsBlocking](#)

CNntpClient::SelectGroup Method

```
INT SelectGroup(  
    LPCTSTR lpszGroupName  
);
```

The **SelectGroup** method selects the specified newsgroup from which articles will be retrieved.

Parameters

lpszGroupName

Pointer to a string which specifies the newsgroup to be selected. This value may be NULL, in which case the current newsgroup is unchanged, but the article count is updated.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method selects the newsgroup and obtains a description and the first and last article numbers for that group.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetGroupName](#), [GetGroupTitle](#), [ListGroup](#)s

CNntpClient::SetLastError Method

```
VOID SetLastError(  
    DWORD dwErrorCode  
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or NNTP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

See Also

[GetErrorString](#), [GetLastError](#), [ShowError](#)

CNntpClient::SetTimeout Method

```
INT SetTimeout(  
    UINT nTimeout  
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[Connect](#), [GetTimeout](#), [IsReadable](#), [IsWritable](#), [Read](#), [Write](#)

CNntpClient::ShowError Method

```
INT ShowError(  
    LPCTSTR lpszAppTitle,  
    UINT uType,  
    DWORD dwErrorCode  
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A pointer to a string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetErrorString](#), [GetLastError](#), [SetLastError](#)

CNntpClient::StoreArticle Method

```
INT StoreArticle(  
    LONG nArticleId,  
    LPCTSTR lpszFileName  
);
```

The **StoreMessage** method retrieves an article from the current newsgroup and stores it in a local file.

Parameters

nArticleId

Number of the article to retrieve. This value must be greater than zero.

lpszFileName

Pointer to a string which specifies the file that the article will be stored in. If the file does not exist, it will be created. If the file does exist, it will be overwritten with the contents of the article.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **StoreArticle** method provides a method of retrieving and storing an article on the local system. The contents of the article is stored as a text file, using the specified file name. This method always causes the caller to block until the entire message has been retrieved, even if the client has been put in asynchronous mode.

If event handling is enabled, the NNTP_EVENT_PROGRESS event will fire periodically during the transfer of the article to the local system. An application can determine how much of the article has been retrieved by calling the **GetTransferStatus** method.

To retrieve the message into a global memory buffer so that it can be passed to the MIME library, use the **GetArticle** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetArticle](#), [GetArticleHeaders](#), [GetTransferStatus](#)

CNntpClient::Write Method

```
INT Write(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Write(  
    LPCTSTR lpszBuffer  
    INT cbBuffer  
);
```

The **Write** method sends the specified number of bytes to the server.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server. In an alternate form of the method, the pointer is to a string.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero, unless a pointer to a string buffer is passed as the parameter. In that case, if the value is -1, all of the characters in the string, up to but not including the terminating null character, will be sent to the server.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is NNTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the next asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableEvents](#), [IsBlocking](#), [IsReadable](#), [IsWritable](#), [Read](#), [RegisterEvent](#)

Network News Transfer Protocol Data Structures

- NEWSARTICLE
- NEWSGROUP
- NNTPTRANSFERSTATUS
- SECURITYCREDENTIALS
- SECURITYINFO
- SYSTEMTIME

NEWSARTICLE Structure

This structure is used by the [GetFirstArticle](#) and [GetNextArticle](#) methods to return information about articles in the currently selected directory.

```
typedef struct _NEWSARTICLE
{
    LONG        nArticleId;
    LONG        nBytes;
    LONG        nLines;
    TCHAR       szSubject[NNTP_MAXSUBJLEN];
    TCHAR       szAuthor[NNTP_MAXAUTHLEN];
    TCHAR       szMessageId[NNTP_MAXMSGIDLEN];
    TCHAR       szReferences[NNTP_MAXREFLEN];
    SYSTEMTIME  stPosted;
} NEWSARTICLE, *LPNEWSARTICLE;
```

Members

nArticleId

A long integer which specifies the article number.

nBytes

The length of the news article in bytes.

nLines

The length of the news article specified as the number of lines of text.

szSubject

A pointer to a string which specifies the subject of the article.

szAuthor

A pointer to a string which specifies the email address of the user who posted the article.

szMessageId

A pointer to a string which specifies the message ID for the article.

szReferences

A pointer to a string which specifies references to the article.

stPosted

A SYSTEMTIME structure which specifies when the article was posted in Universal Coordinated Time (UTC).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csnews10.lib

Unicode: Implemented as Unicode and ANSI versions.

NEWSGROUP Structure

This structure is used by the [GetFirstGroup](#) and [GetNextGroup](#) methods to return information about available newsgroups.

```
typedef struct _NEWSGROUP
{
    LONG        nFirstArticle;
    LONG        nLastArticle;
    DWORD       dwAccess;
    TCHAR       szName[NNTP_MAXGRPNAMELEN];
} NEWSGROUP, *LPNEWSGROUP;
```

Members

nFirstArticle

A long integer which specifies the article number of the first available article in the newsgroup.

nLastArticle

A long integer which specifies the article number of the last available article in the newsgroup. Note that posted articles may not be contiguous in the range between the first and last article numbers. Some servers may assign numbers in a different order than the articles were posted, or there may be gaps where articles have been removed.

dwAccess

An unsigned integer which specifies the access mode for the group. It may be one of the following values:

Constant	Description
NNTP_GROUP_READONLY	The group is read-only and cannot be modified. Attempts to post articles to the newsgroup will result in an error.
NNTP_GROUP_READWRITE	Articles can be posted to the newsgroup. Even though a newsgroup is read-write, it may require that the client authenticate before being given permission to post articles to the server.
NNTP_GROUP_MODERATED	The newsgroup is moderated and articles can only be posted by the group moderator. To request that an article be posted to the newsgroup, you must email the message to the moderator.

szName

A pointer to a string which specifies the name of the newsgroup.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csnews10.lib

Unicode: Implemented as Unicode and ANSI versions.

NNTPTTRANSFERSTATUS Structure

This structure is used by the [GetTransferStatus](#) method to return information about an article transfer in progress.

```
typedef struct _NNTPTTRANSFERSTATUS
{
    UINT    nArticleId;
    DWORD   dwBytesTotal;
    DWORD   dwBytesCopied;
    DWORD   dwBytesPerSecond;
    DWORD   dwTimeElapsed;
    DWORD   dwTimeEstimated;
} NNTPTTRANSFERSTATUS, *LPNNTPTTRANSFERSTATUS;
```

Members

nArticleId

The article ID of the current article that is being transferred. If an article is being posted, this member will be set to zero.

dwBytesTotal

The total number of bytes that will be transferred. If the article is being copied from the server to the local host, this is the size of the article on the server. If the article is being posted to the server, it is the size of article on the local system. If the article size cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the transfer started.

dwTimeEstimated

The estimated number of seconds until the transfer is completed. This is based on the average number of bytes transferred per second.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

SECURITYCREDENTIALS Structure

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on

	what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that

will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A pointer to a null terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A pointer to a null terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A pointer to a null terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A pointer to a null terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
------------	-------------

CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
----	---

MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
----	--

ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.
------	--

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

A pointer to a null terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used for SSH connections

and should always be NULL when establishing a secure connection using SSL or TLS.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* member should be set to CREDENTIAL_STORE_FILENAME and the *lpzCertStore* member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS#12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the *lpzPassword* member or the library will be unable to access the certificate.

Note that the *lpzUserName* and *lpzPassword* members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

The TLS 1.1 and TLS 1.2 protocols are only supported on Windows 7, Windows Server 2008 R2 and later versions of the platform. If these options are specified and the application is running on Windows XP or Windows Vista, the protocol version will be downgraded to TLS 1.0 for backwards compatibility with those platforms.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

SECURITYINFO Structure

This structure contains information about a secure connection that has been established.

```
typedef struct _SECURITYINFO
{
    DWORD        dwSize;
    DWORD        dwProtocol;
    DWORD        dwCipher;
    DWORD        dwCipherStrength;
    DWORD        dwHash;
    DWORD        dwHashStrength;
    DWORD        dwKeyExchange;
    DWORD        dwCertStatus;
    SYSTEMTIME   stCertIssued;
    SYSTEMTIME   stCertExpires;
    LPCTSTR      lpszCertIssuer;
    LPCTSTR      lpszCertSubject;
    LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to **sizeof(SECURITYINFO)** prior to passing the address of this structure to the function. Note that if this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the protocol that was selected to establish the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The

	<p>correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.</p>
SECURITY_PROTOCOL_TLS10	<p>The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS11	<p>The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS12	<p>The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.</p>

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_CIPHER_RC2	The RC2 block cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
SECURITY_CIPHER_DES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher, using 56-bit

	keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher (also known as the Rijndael cipher) is a fixed block size cipher which use a key size of 128, 192 or 256 bits. This cipher is supported on Windows XP SP3 and later versions of the operating system.
SECURITY_CIPHER_SKIPJACK	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
SECURITY_CIPHER_BLOWFISH	The Blowfish block cipher was selected. This is a variable key length cipher up to 448 bits, using a 64-bit block size.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected. One of the following values will be returned:

Constant	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected. One of the following values will be returned:

--	--

Constant	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography. This key exchange algorithm is only supported on Windows XP SP3 and later versions of the operating system.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server. This member only has meaning for connections using the SSL or TLS protocols. One of the following values will be returned:

Constant	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertIssuer

A pointer to a string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertSubject

A pointer to a string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszFingerprint

A pointer to a string which contains a sequence of hexadecimal values that uniquely identify the server. This member is only used when a connection has been established using the Secure Shell (SSH) protocol.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Unicode: Implemented as Unicode and ANSI versions.

SYSTEMTIME Structure

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include windows.h.

News Feed Class Library

Retrieve and process the contents of a syndicated news feed.

Reference

- [Class Methods](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

Class Name	CNewsFeed
File Name	CSRSSV10.DLL
Version	10.0.1468.2518
LibID	1F65A283-7BC8-4F5B-B739-D211EAF1CA35
Import Library	CSRSSV10.LIB
Dependencies	None

Overview

Really Simple Syndication (RSS) is a collection of standardized formats that are used to publish information about content that is frequently changed. A news feed is published in XML format, which contains one or more items that includes summary text, hyperlinks to source content and additional metadata that is used to describe the item. News feeds can be used for a variety of purposes, including providing updates for weblogs, news headlines, video and audio content. RSS can also be used for other purposes, such as a software updates, where new updates are listed as items in the feed.

News feeds can be accessed remotely from a web server, or locally as an XML formatted text file. The source of the feed is determined by the URI scheme that is specified. If the http or https scheme is specified, then the feed is retrieved from a web server. If the file scheme is used, the feed is considered to be local and is accessed from the disk or local network. The CNewsFeed class provides an interface that enables you to open a feed by URL and iterate through each of the items in the feed or search for a specific feed item. The class also provides a method that can be used to parse a string that contains XML data in RSS format, where the feed may have been retrieved from other sources such as a database.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-

bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

News Feed Class Methods

Class	Description
CNewsFeed	Constructor which initializes the current instance of the class
~CNewsFeed	Destructor which releases resources allocated by the class
Method	Description
AttachHandle	Attach the specified client handle to this instance of the class
CloseFeed	Close the specified news feed and release memory allocated for the channel
DetachHandle	Detach the handle for the current instance of this class
DisableTrace	Disable logging of network function calls
EnableTrace	Enable logging of network function calls to a text file
FindItem	Find a specific news feed item using its unique identifier (GUID) property
GetErrorString	Return a description for the specified error code
GetFirstItem	Return information about the first item in the news feed channel
GetHandle	Return the client handle used by this instance of the class
GetItem	Return information about the specified news feed item
GetItemCount	Return the number of news feed items in the channel
GetItemProperty	Return the value of the specified news feed item property or attribute
GetItemText	Return the text description of the specified news feed item
GetLastError	Return the last error code
GetNextItem	Return information about the next item in the news feed channel
IsInitialized	Determine if the class has been successfully initialized
OpenFeed	Open the specified news feed and return information about the channel
ParseFeed	Parse the contents of a string and return information about the channel
RefreshFeed	Refresh the specified news feed, updating the items in the channel
SetLastError	Set the last error code
ShowError	Display a message box with a description of the specified error
StoreFeed	Store the contents of the specified news feed in an XML formatted text file
ValidateFeed	Validate the contents of the specified news feed, returning the number of items in the feed

CNewsFeed::CNewsFeed Method

`CNewsFeed()`;

The **CNewsFeed** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **RssInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csrsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[~CNewsFeed, IsInitialized](#)

CNewsFeed::~CNewsFeed

`~CNewsFeed();`

The **CNewsFeed** destructor releases resources allocated by the current instance of the **CNewsFeed** object. It also uninitialized the library if there are no other concurrent uses of the class.

Remarks

When a **CNewsFeed** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csrsv10.lib`

See Also

[CNewsFeed](#)

CNewsFeed::AttachHandle Method

```
VOID AttachHandle(  
    HCHANNEL hChannel  
);
```

The **AttachHandle** method attaches the specified handle to the current instance of the class.

Parameters

hChannel

The handle to the channel that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a handle created outside of the class using the SocketTools API. Once the handle is attached to the class, the other class member functions may be used with that news feed channel.

If a handle already has been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous news feed channel from being closed, you must call the **DetachHandle** method. Failure to close the detached handle may result in a resource leak in your application.

Note that the *hChannel* parameter is presumed to be valid and no checks are performed to ensure its validity. Specifying an invalid handle value will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrssv10.lib

See Also

[DetachHandle](#), [GetHandle](#)

CNewsFeed::CloseFeed Method

```
BOOL CloseFeed();
```

The **CloseFeed** method closes the specified news feed and releases memory allocated for the channel.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **CloseFeed** method must be called whenever the application has completed processing the news feed. It is important to note that the memory allocated for the channel will be released when this function is called, which means that any data referenced in the `RSSCHANNEL` and `RSSCHANNELITEM` structures will no longer be valid and must not be used by the application after the feed has been closed.

This method can fail if the feed is currently being updated, such as when the **RefreshFeed** method is called. In this case, the channel handle will not be released and the application must attempt to close the feed at a later time.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csrsv10.lib`

See Also

[OpenFeed](#) [ParseFeed](#) [StoreFeed](#)

CNewsFeed::DetachHandle Method

```
HCHANNEL DetachHandle();
```

The **DetachHandle** method detaches the handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the handle associated with the current instance of the class object. If there is no open news feed, the value `INVALID_CHANNEL` will be returned.

Remarks

This method is used to detach a handle created by the class for use with the SocketTools API. Once the handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly closed at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csrsv10.lib`

See Also

[AttachHandle](#), [GetHandle](#)

CNewsFeed::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrssv10.lib

See Also

[EnableTrace](#)

CNewsFeed::EnableTrace Method

```
BOOL EnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **EnableTrace** method enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace method logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableTrace](#)

CNewsFeed::FindItem Method

```
BOOL FindItem(  
    LPCTSTR lpszValue,  
    DWORD dwOptions,  
    LPRSSCHANNELITEM lpItem  
);
```

```
BOOL FindItem(  
    LPCTSTR lpszValue,  
    LPRSSCHANNELITEM lpItem  
);
```

The **FindItem** method searches for an item in the news feed channel which matches the unique identifier (GUID) value and returns information about that item.

Parameters

lpszValue

A pointer to a string which specifies the value of the item being searched for. This value should uniquely identify the item in the feed, and this parameter cannot be an empty string or NULL pointer.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
RSS_FIND_GUID	Search the feed for items with a matching GUID property value. This is the default option, and is the only item property that is guaranteed to be unique in the feed. The search is case-sensitive, requiring that the <i>lpszValue</i> parameter match the property value exactly.
RSS_FIND_LINK	Search the feed for items with a matching link property value. For feeds that do not specify a GUID property, this is the recommended option for searching for an item. The search is not case-sensitive.
RSS_FIND_TITLE	Search the feed for items with a matching title. This option should not be used if you must ensure that the item returned is unique in the feed because there may be multiple items with the same title in the feed. The search is not case-sensitive.
RSS_FIND_PUBDATE	Search the feed for items with a matching publishing date. This option should not be used if you must ensure that the item returned is unique in the feed because more than one item may have the same publishing date. The format of the date string must match the standard format used with the RSS protocol and the match is not case-sensitive.

lpItem

A pointer to a **RSSCHANNELITEM** structure which contains information about the specified item in the news feed. This structure is initialized by the method and the parameter can never

be specified as a NULL pointer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

It is recommended that you use the `RSS_FIND_GUID` option with news feeds that are using version 2.0 or later of the RSS specification. If the feed uses an earlier version, items may not include a GUID property. It is also possible that a feed may omit the GUID property even though it is considered a requirement for the current RSS specification. For the broadest compatibility with all news feeds, an application should not depend on being able to search for a specific news feed item by its GUID.

Only the GUID property is guaranteed to be unique in the feed. If the feed does not specify GUIDs for the news items, the application must use an alternate criteria such as the item hyperlink or publishing date. If there are multiple items that match the *lpzValue* value, the first matching item will be returned.

The data referenced in the **RSSCHANNELITEM** structure should be considered read-only and never modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed, the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csrsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFirstItem](#), [GetItem](#), [GetItemProperty](#), [GetItemText](#), [GetNextItem](#), [RSSCHANNELITEM](#)

CNewsFeed::GetErrorString Method

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);  
  
INT GetErrorString(  
    DWORD dwErrorCode,  
    CString& strDescription  
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csrssv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLastError](#), [SetLastError](#), [ShowError](#)

CNewsFeed::GetFirstItem Method

```
BOOL GetFirstItem(  
    LPRSSCHANNELITEM LpItem  
);
```

The **GetFirstItem** method returns information about the first item in the news feed channel.

Parameters

lpItem

A pointer to a **RSSCHANNELITEM** structure which contains information about the first item in the news feed. This structure is initialized by the method and the parameter can never be specified as a NULL pointer.

Return Value

If the method succeeds, it returns a non-zero value. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetFirstItem** method is used in conjunction with the **GetNextItem** method to enumerate the available items in the specified news feed channel. If this method fails, it typically indicates that the channel does not contain any valid news items or that the format of the news feed is invalid.

The data referenced in the **RSSCHANNELITEM** structure should be considered read-only and never modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed, the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: csrssv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FindItem](#), [GetItem](#), [GetItemProperty](#), [GetItemText](#), [GetNextItem](#), [RSSCHANNELITEM](#)

CNewsFeed::GetHandle Method

```
HCHANNEL GetHandle();
```

The **GetHandle** method returns the handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the handle associated with the current instance of the class object. If there is no open news feed, the value `INVALID_CHANNEL` will be returned.

Remarks

This method is used to obtain the handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csrsv10.lib`

See Also

[AttachHandle](#), [DetachHandle](#), [IsInitialized](#)

CNewsFeed::GetItem Method

```
BOOL GetItem(  
    UINT nItemId,  
    LPRSSCHANNELITEM lpItem  
);
```

The **GetItem** method returns information about the specified item in the news feed channel.

Parameters

nItemId

An integer value which identifies the news feed item. The first item identifier in the news feed has a value of one, and that value is incremented for each additional item in the feed. If this parameter is zero or specifies a value larger than the number of items in the feed, this method will fail.

lpItem

A pointer to a **RSSCHANNELITEM** structure which contains information about the specified item in the news feed. This structure is initialized by the method and the parameter can never be specified as a NULL pointer.

Return Value

If the method succeeds, it returns a non-zero value. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetItem** method is used to return information about a specific item in the news feed. If this method fails, it typically indicates that the item ID is invalid or that the feed does not contain any valid news items. The **GetItemCount** method can be used to determine the number of items contained in the feed channel.

The data referenced in the **RSSCHANNELITEM** structure should be considered read-only and never modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed, the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csrssv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FindItem](#), [GetFirstItem](#), [GetItemCount](#), [GetItemProperty](#), [GetItemText](#), [GetNextItem](#), [RSSCHANNELITEM](#)

CNewsFeed::GetItemCount Method

```
INT GetItemCount();
```

The **GetItemCount** method returns the number of items in the news feed channel.

Parameters

None.

Return Value

If the method succeeds, the return value is the number of items in the news feed. A value of zero indicates that the feed channel is empty. If the method fails, the return value is `RSS_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **GetItemCount** method is used to determine the number of items that are contained in the news feed channel, and therefore determine the maximum value of the item identifier which can be used to reference a specific item in the feed. This value is the same as the value specified by the *nItemCount* member of the **RSSCHANNEL** structure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csrsv10.lib`

See Also

[FindItem](#), [GetItem](#), [GetItemProperty](#), [GetItemText](#), [RSSCHANNEL](#)

CNewsFeed::GetItemProperty Method

```
INT GetItemProperty(  
    UINT nItemId,  
    LPCTSTR lpszProperty,  
    LPCTSTR lpszAttribute,  
    LPTSTR lpszValue,  
    INT nMaxLength  
);
```

```
INT GetItemProperty(  
    UINT nItemId,  
    LPCTSTR lpszProperty,  
    LPCTSTR lpszAttribute,  
    CString& strValue  
);
```

The **GetItemProperty** method is used to return the value of a property for the specified item in the news feed channel.

Parameters

nItemId

An integer value which identifies the news feed item. The first item identifier in the news feed has a value of one, and that value is incremented for each additional item in the feed. If this parameter is zero or specifies a value larger than the number of items in the feed, this method will fail.

lpszProperty

A pointer to a string which specifies the name of the item property. This parameter cannot point to an empty string or specify a NULL pointer.

lpszAttribute

A pointer to a string which specifies the name of an attribute for the property. If this parameter is an empty string or NULL pointer, the method will return the value of the property, rather than an attribute of the specified property.

lpszValue

A pointer to a string buffer which will contain the value of the specified item property or attribute. This string should be large enough to contain the property value. If this parameter is a NULL pointer, it will be ignored and the method will only return the length of value for the specified property.

nMaxLength

The maximum number of characters that may be copied into the property value buffer. If the value of this parameter is zero, then the *lpszValue* parameter is ignored and the method will only return the length of the value for the specified property.

Return Value

If the method succeeds, the return value is the length of the property value string. A return value of zero indicates that the property does not contain any value. If the method fails, the return value is `RSS_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **GetItemProperty** method is primarily used with custom item properties that may be used

with extensions to the news feed. The standard properties for an item such as the title, link and description can be obtained using **GetItem** and related methods. However, if items in the feed contain custom properties that are not part of the standard RSS format, this method can be used to obtain those values.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FindItem](#), [GetItem](#), [GetItemText](#), [RSSCHANNELITEM](#)

CNewsFeed::GetItemText Method

```
INT GetItemText(  
    UINT nItemId,  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);
```

```
INT GetItemText(  
    UINT nItemId,  
    CString& strBuffer  
);
```

The **GetItemText** method is used to return a copy of an item's description.

Parameters

nItemId

An integer value which identifies the news feed item. The first item identifier in the news feed has a value of one, and that value is incremented for each additional item in the feed. If this parameter is zero or specifies a value larger than the number of items in the feed, this method will fail.

lpszBuffer

A pointer to a string buffer which will contain the value of the item description. If this parameter is a NULL pointer, it will be ignored and the method will only return the length of the item description.

nMaxLength

The maximum number of characters that may be copied into the description buffer. If the value of this parameter is zero, then the *lpszValue* parameter is ignored and the method will only return the length of the item description.

Return Value

If the method succeeds, the return value is the length of the item description. A return value of zero indicates that the item does not have a description. If the method fails, the return value is `RSS_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **GetItemText** function is used to obtain a copy of the string that describes the specified item. Typically this is text that provides a summary of the news feed item and is used in conjunction with the item's title and hyperlink to additional content.

The content of an item description is typically either plain text or HTML formatted text. It is the responsibility of the application to display the content in a format is appropriate for the end-user.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csrsvg10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FindItem](#), [GetItem](#), [GetItemProperty](#), [RSSCHANNELITEM](#)

CNewsFeed::GetLastError Method

```
DWORD GetLastError();  
  
DWORD GetLastError(  
    CString& strDescription  
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError(0)** when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or RSS_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csrsv10.lib

See Also

[GetErrorString](#), [SetLastError](#), [ShowError](#)

CNewsFeed::GetNextItem Method

```
BOOL GetNextItem(  
    LPRSSCHANNELITEM LpItem  
);
```

The **GetNextItem** method returns information about the next item in the news feed channel.

Parameters

LpItem

A pointer to a **RSSCHANNELITEM** structure which contains information about the next item in the news feed. This structure is initialized by the method and the parameter can never be specified as a NULL pointer.

Return Value

If the method succeeds, it returns a non-zero value. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetNextItem** method is used in conjunction with the **GetFirstItem** method to enumerate the available items in the specified news feed channel. If this method fails, it typically indicates that there are no more items in the news feed channel.

The data referenced in the **RSSCHANNELITEM** structure should be considered read-only and never modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed, the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrssv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FindItem](#), [GetFirstItem](#), [GetItem](#), [GetItemProperty](#), [GetItemText](#), [RSSCHANNELITEM](#)

CNewsFeed::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrssv10.lib

See Also

[CNewsFeed](#)

CNewsFeed::OpenFeed Method

```
BOOL OpenFeed(  
    LPCTSTR lpszFeedUrl,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPRSSCHANNEL lpChannel  
);
```

```
BOOL OpenFeed(  
    LPCTSTR lpszFeedUrl,  
    UINT nTimeout,  
    LPRSSCHANNEL lpChannel  
);
```

```
BOOL OpenFeed(  
    LPCTSTR lpszFeedUrl,  
    LPRSSCHANNEL lpChannel  
);
```

The **OpenFeed** method is used to open a news feed and return a handle which can be used to access the individual news items in the feed.

Parameters

lpszFeedUrl

A pointer to a string which specifies the URL for the news feed. To access a news feed on a web server, a standard http or https URL may be specified. To access a file on the local system or network share, a file name or file URL may be specified.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation. This parameter is ignored if the *lpszFeedUrl* parameter specifies a local file name or URL.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
RSS_OPTION_NONE	No additional options are specified and the news feed is processed using relaxed rules when checking the validity of the feed. The library will attempt to automatically compensate for a feed that is malformed or does not strictly conform to the RSS standard.
RSS_OPTION_STRICT	The news feed content should be processed using strict rules to ensure that the feed meets the appropriate RSS standard specification and all feed property values are case-sensitive. By default, relaxed rules are used which allows the application to open a feed that may not strictly conform to the standard specification.

lpChannel

A pointer to an **RSSCHANNEL** structure which contains information about the news feed

channel such as the feed title, hyperlink and description. If the parameter is not NULL, the structure is initialized by the method. If the parameter is NULL, it is ignored and no information is returned.

Return Value

If the method succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

A news feed may be local or remote, depending on the URL that is specified. If a local file name or file URL is specified for the feed, then it is opened locally and no network access is required. If an http or https URL is specified, then **OpenFeed** will attempt to download the feed from the server and store it temporarily on the local system. Accessing a remote feed requires that the application has permission to establish a connection with the server and will cause the application to block until the feed has been downloaded, the operation times out or an error occurs.

Although the **OpenFeed** method will meet the needs of most applications, if you require more complex functionality such as retrieving the feed asynchronously in the background or event notifications for large transfers, you can use the SocketTools [Hypertext Transfer Protocol](#) API to download the news feed and then use the **ParseFeed** method to parse the contents.

The data referenced in the **RSSCHANNEL** structure should be considered read-only and never modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed, the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CloseFeed](#), [ParseFeed](#), [RefreshFeed](#), [StoreFeed](#), [RSSCHANNEL](#)

CNewsFeed::ParseFeed Method

```
BOOL ParseFeed(  
    LPCTSTR lpszFeedXml,  
    DWORD dwOptions,  
    LPRSSCHANNEL lpChannel  
);
```

```
BOOL ParseFeed(  
    LPCTSTR lpszFeedXml,  
    LPRSSCHANNEL lpChannel  
);
```

The **ParseFeed** method is used to parse the contents of a news feed, returning a handle which can be used to access the individual news items in the feed.

Parameters

lpszFeedXml

A pointer to a string which contains the contents of the news feed. The string must contain XML formatted data that conforms to the RSS standard specification. This parameter cannot specify an empty string or a NULL pointer.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
RSS_OPTION_NONE	No additional options are specified.
RSS_OPTION_STRICT	The news feed content should be processed using strict rules to ensure that the feed meets the appropriate RSS standard specification and all feed property values are case-sensitive. By default, relaxed rules are used which allows the application to open a feed that may not strictly conform to the standard specification.

lpChannel

A pointer to an **RSSCHANNEL** structure which contains information about the news feed channel such as the feed title, hyperlink and description. If the parameter is not NULL, the structure is initialized by the method. If the parameter is NULL, it is ignored and no information is returned.

Return Value

If the method succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ParseFeed** method is an alternative to the **OpenFeed** method, enabling the application to process a news feed from alternative sources such as a database or compressed file. It is important to note that the string which contains the news feed XML must be properly formatted and conform to the RSS standard specification.

The data referenced in the **RSSCHANNEL** structure should be considered read-only and never

modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed, the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrssv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CloseFeed](#), [OpenFeed](#), [RefreshFeed](#), [StoreFeed](#), [RSSCHANNEL](#)

CNewsFeed::RefreshFeed Method

```
BOOL RefreshFeed(  
    LPRSSCHANNEL lpChannel  
);
```

The **RefreshFeed** method reloads the news feed and updates the items in the channel.

Parameters

lpChannel

A pointer to an **RSSCHANNEL** structure which contains information about the news feed channel such as the feed title, hyperlink and description. If the parameter is not NULL, the structure is initialized by the method. If the parameter is NULL, it is ignored and no information is returned.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

When the **RefreshFeed** method is called, the news feed is reloaded from the original source and the items in the channel are updated. For news feeds that are frequently updated, the *nTimeToLive* member of the **RSSCHANNEL** structure can provide a hint to the application as to how frequently the feed should be refreshed.

If the news feed was originally opened using an http or https URL, this method will download the updated feed from the server and store it temporarily on the local system. Accessing a remote feed requires that the application has permission to establish a connection with the server and will cause the application to block until the feed has been downloaded, the operation times out or an error occurs. The same timeout period and options will be used as when the feed was originally opened.

The **RefreshFeed** method should only be used if the feed was opened using the **OpenFeed** method, otherwise the method will fail with an error indicating that the operation is not supported.

The data referenced in the **RSSCHANNEL** structure should be considered read-only and never modified by the application. The members of this structure returned by previous calls to either the **OpenFeed** or **RefreshFeed** methods will no longer be valid and should not be referenced. Likewise, the members of an **RSSCHANNELITEM** structure will no longer be valid after this method returns.

It is important that the application does not make any assumptions about the number of news items in the channel, or the content associated with those items after the **RefreshFeed** function has been called. For example, never assume that the number of items in the channel remains the same, or that the item IDs for each item remains the same. If you need to find a specific item in the news feed, use the **FindItem** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrssv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CloseFeed](#), [FindItem](#), [GetItem](#), [OpenFeed](#), [StoreFeed](#), [RSSCHANNEL](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CNewsFeed::SetLastError Method

```
VOID SetLastError(  
    DWORD dwErrorCode  
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or RSS_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csrssv10.lib

See Also

[GetErrorString](#), [GetLastError](#), [ShowError](#)

CNewsFeed::ShowError Method

```
INT ShowError(  
    LPCTSTR lpszAppTitle,  
    UINT uType,  
    DWORD dwErrorCode  
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A pointer to a string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csrsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetErrorString](#), [GetLastError](#), [SetLastError](#)

CNewsFeed::StoreFeed Method

```
BOOL StoreFeed(  
    LPCTSTR lpszFileName  
);
```

The **StoreFeed** method stores the contents of the news feed in an XML formatted text file.

Parameters

lpszFileName

A pointer to a string which specifies the name of the file on the local system. The contents of the news feed will be stored in this file. If the file does not exist, it will be created; otherwise it will overwrite the contents of the file.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[OpenFeed](#), [ParseFeed](#)

CNewsFeed::ValidateFeed Method

```
INT ValidateFeed(  
    LPCTSTR lpszFeedUrl,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSYSTEMTIME lpstModified  
);
```

```
INT ValidateFeed(  
    LPCTSTR lpszFeedUrl,  
    LPSYSTEMTIME lpstModified  
);
```

```
INT ValidateFeed(  
    LPCTSTR lpszFeedUrl  
);
```

The **RssValidateFeed** method is used to validate a news feed, returning the number of items in the feed and the date it was last modified.

Parameters

lpszFeedUrl

A pointer to a string which specifies the URL for the news feed. To access a news feed on a web server, a standard http or https URL may be specified. To access a file on the local system or network share, a file name or file URL may be specified.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation. This parameter is ignored if the *lpszFeedUrl* parameter specifies a local file name or URL.

dwOptions

An unsigned integer that specifies one or more options. This parameter is reserved for future use and should always have a value of zero.

lpstModified

A pointer to a **SYSTEMTIME** structure which will specify the date that the feed was last modified when the method returns. If the parameter is NULL, it is ignored and no information is returned.

Return Value

If the method succeeds, the return value is the number of items in the news feed channel. If the method fails, the return value is **RSS_ERROR**. To get extended error information, call **RssGetLastError**.

Remarks

The **RssValidateFeed** method can be used to check that a news feed exists and is properly formatted. If the contents of the feed are valid, the method will return the number of items in the feed and the date that it was last modified. This can be useful for applications that want to periodically check a news feed and determine if the contents have changed.

The **SYSTEMTIME** structure that is populated by the method specifies the date when the feed was last modified. The method first checks the value of the `lastBuildDate` property of the feed channel. If that property is not defined, then it will use the value of the `pubDate` property. If neither are defined, then the structure members will have a value of zero.

The validation process imposes strict checks on the structure of the news feed and requires that it conform to the RSS specification. For example, the feed must have a title, link and description. Each item in the feed must have either a title or description, and the hyperlinks specified in the feed must be valid. If the feed XML is malformed, or a required property of the feed is invalid or missing, this method will fail.

If a news feed cannot be validated, it still may be possible to open the feed using the **OpenFeed** method. By default, relaxed rules are used when parsing the contents of the feed and it does not check to ensure all required properties are defined and have valid values.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RssOpenFeed](#), [RssParseFeed](#), [RssRefreshFeed](#), [RssStoreFeed](#), [SYSTEMTIME](#)

News Feed Data Structures

- RSSCHANNEL
- RSSCHANNELITEM
- SYSTEMTIME

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

RSSCHANNEL Structure

This structure contains information about the properties of the news feed channel.

```
typedef struct _RSSCHANNEL
{
    UINT        nItemCount;
    UINT        nTimeToLive;
    WORD        wVersionMajor;
    WORD        wVersionMinor;
    DWORD       dwFlags;
    DWORD       dwReserved;
    LPCTSTR     lpszTitle;
    LPCTSTR     lpszLink;
    LPCTSTR     lpszDescription;
    LPCTSTR     lpszCategory;
    LPCTSTR     lpszLanguage;
    LPCTSTR     lpszCopyright;
    LPCTSTR     lpszEditor;
    LPCTSTR     lpszWebmaster;
    LPCTSTR     lpszGenerator;
    LPCTSTR     lpszImageLink;
    LPCTSTR     lpszImageTitle;
    LPCTSTR     lpszImageUrl;
    SYSTEMTIME  stPublished;
    SYSTEMTIME  stLastBuild;
} RSSCHANNEL, *LPRSSCHANNEL;
```

Members

nItemCount

An integer value which specifies number of news items in the channel.

nTimeToLive

An integer value which specifies the frequency in seconds at which the feed should be refreshed to obtain updated information. Not all feeds specify a time-to-live, in which case this member will have a value of zero.

wVersionMajor

A word value which specifies the major version number for the news feed.

wVersionMinor

A word value which specifies the minor version number for the news feed.

dwFlags

A value which specifies one or more option flags for the news feed channel. Currently there are no option flags defined and this member is reserved for future expansion.

dwReserved

A value reserved for future expansion.

lpszTitle

A pointer to a string which specifies the name of the channel. If the content of the news feed corresponds to a website, this is typically the same as the title of the website. If a title has not been specified, this member will be NULL. Note that a strictly conforming news feed requires a title.

lpszLink

A pointer to a string which specifies a URL to the website corresponding to the channel. Note that this is not the URL of the news feed itself. Typically it is a link to the home page of the site which owns the news feed. If a link has not been specified, this member will be NULL. Note that strictly conforming news feed requires a link.

lpszDescription

A pointer to a string which describes the channel. This provides an overview of the news feed and the type of information that is provided. If a description of the feed has not been specified, this member will be NULL. Note that a strictly conforming news feed requires a description.

lpszCategory

A pointer to a string which defines the category or categories that the channel belongs to. This property is optional and the category names themselves are user-defined. If a category has not been specified, this member will be NULL.

lpszLanguage

A pointer to a string which defines the language the channel is written in, using the standard language codes. This property is optional and if this member is NULL, the English language is typically presumed to be the default.

lpszCopyright

A pointer to a string which specifies a copyright notice for the content. If a copyright has not been specified, this member will be NULL.

lpszEditor

A pointer to a string which identifies the person responsible for managing the content of the news feed. If this property is defined, it is typically the name and email address of the feed editor. If an editor has not been specified, this member will be NULL.

lpszWebmaster

A pointer to a string which identifies the person responsible for technical issues related to the news feed. If this property is defined, it is typically the name and email address of a system administrator. If a webmaster has not been specified, this member will be NULL.

lpszGenerator

A pointer to a string which identifies the application that was used to create the news feed. If the application that generated the feed has not been specified, this member will be NULL.

lpszImageLink

A pointer to a string which specifies a URL to the website corresponding to the channel. In most cases, this is the same URL that is specified by the *lpszLink* member. If an image link has not been specified, this member will be NULL.

lpszImageTitle

A pointer to a string which identifies the image associated with the channel. This is usually a brief description of the image, and may be the same as the value specified by the *lpszTitle* member. If an image title has not been specified, this member will be NULL.

lpszImageUrl

A pointer to a string which specifies a URL for the image associated with the channel. An application can download this image and display it with the contents of the news feed. If an image URL has not been specified, this member will be NULL.

stPublished

The date that the news feed was published. For example, a feed that is associated with a weekly print publication may update this value once per week. Note that this is not necessarily the date

that the feed was last modified. If the channel does not specify the publish date, this structure will contain all zeroes.

stLastBuild

The date that the content of the channel was last modified. If the channel does not specify the build date, this structure will contain all zeroes.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

RSSCHANNELITEM Structure

This structure contains information about the properties of an item in a news feed channel.

```
typedef struct _RSSCHANNELITEM
{
    UINT            nItemId;
    DWORD          dwFlags;
    DWORD          dwReserved;
    LPCTSTR        lpszTitle;
    LPCTSTR        lpszLink;
    LPCTSTR        lpszText;
    LPCTSTR        lpszGuid;
    LPCTSTR        lpszAuthor;
    LPCTSTR        lpszSource;
    LPCTSTR        lpszComments;
    LPCTSTR        lpszEnclosure;
    SYSTEMTIME     stPublished;
} RSSCHANNELITEM, *LPRSSCHANNELITEM;
```

Members

nItemId

An integer which identifies this item in the channel.

dwFlags

A value which specifies one or more option flags for the item. Currently there are no option flags defined and this member is reserved for future expansion.

dwReserved

A value reserved for future expansion.

lpszTitle

A pointer to a string which specifies the title of the item. If a title for the item has not been specified, this member will be NULL.

lpszLink

A pointer to a string which specifies a URL that typically links to additional information related to the item. If a link for the item has not been specified, this member will be NULL.

lpszText

A pointer to a string which specifies a summary or description of the item. This may contain either plain text or HTML formatted text, and there is no fixed limit to the length of the text. If no text has been specified for the item, this member will be NULL.

lpszGuid

A pointer to a string which uniquely identifies the item in the channel. If this property is defined, it is guaranteed to be a unique, persistent value. It is important to note that this string does not have to be a standard GUID reference number, it can be any unique string. In many cases it is the same value as the item hyperlink specified by the *lpszLink* member, although an application should never depend on this behavior. If there is no unique identifier associated with the item, this member will be NULL.

lpszAuthor

A pointer to a string which identifies the author of the item. If this property is defined, it is typically the name and email address of the person who created the content that the item links to. If the author is not specified, this member will be NULL.

lpszSource

A pointer to a string which identifies the source of the item, specified as a URL for the original news feed that contained it. This typically used to propagate credit for items that are aggregated by a third-party and re-published in their own channel. If the source is not specified, this member will be NULL.

lpszComments

A pointer to a string which specifies a URL that links to further discussion about the item. Typically this is a link to the comment area of a weblog or a forum topic specific to the item. If a comment link is not specified, this member will be NULL.

lpszEnclosure

A pointer to a string which specifies a URL that links to a file related to the item. This is similar to an attachment in an email message, however instead of the item containing the contents of the attached file, it only specifies a link to the file. Enclosures are most commonly used with podcasting where an item is linked to an audio or video file, however the link may reference any type of file. If there is no enclosure specified for the item, this member will be NULL.

stPublished

The date that the item was published. If the item does not specify the publish date, this structure will contain all zeroes.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

SYSTEMTIME Structure

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include windows.h.

Post Office Protocol Class Library

List and retrieve email messages from a mail server.

Reference

- [Class Methods](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

Class Name	CPopClient
File Name	CSPOPV10.DLL
Version	10.0.1468.2518
LibID	E4B30E06-4091-4A79-8A4C-17547C76A995
Import Library	CSPOPV10.LIB
Dependencies	None
Standards	RFC 1939

Overview

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

This library supports secure connections using the standard SSL and TLS protocols. Both implicit and explicit SSL connections can be established, enabling the library to work with a wide variety of servers.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Post Office Protocol Class Methods

Class	Description
CPopClient	Constructor which initializes the current instance of the class
~CPopClient	Destructor which releases resources allocated by the class
Method	Description
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified client handle to another thread
Cancel	Cancel the current blocking operation
ChangePassword	Change the specified mail account password
Command	Send a command to the server
Connect	Connect to the specified server
CreateSecurityCredentials	Allocate a structure to establish client security credentials
DeleteMessage	Delete the specified message from the mailbox
DeleteSecurityCredentials	Delete the specified client security credentials
DetachHandle	Detach the handle for the current instance of this class
DisableEvents	Disable the event notification mechanism
DisableTrace	Disable logging of socket function calls to the trace log
Disconnect	Disconnect from the current server
EnableEvents	Enable the client event notification mechanism
EnableTrace	Enable logging of socket function calls to a file
FreezeEvents	Suspend and resume event handling by the client
GetErrorString	Return a description for the specified error code
GetHandle	Return the client handle used by this instance of the class
GetHeaderValue	Return the value of the specified header field
GetLastError	Return the last error code
GetMessage	Retrieve the specified message from the server
GetMessageCount	Return the number of messages available in the mailbox
GetMessageHeaders	Retrieve the specified message header from the server
GetMessageId	Return the message ID string for the specified message
GetMessageSender	Return the address of the message sender
GetMessageSize	Return the size of the specified message
GetMessageUid	Return the unique identifier for the specified message
GetResultCode	Return the result code from the previous command

GetResultString	Return the result string from the previous command
GetSecurityInformation	Return security information about the current client connection
GetStatus	Return the current status of the client
GetTimeout	Return the number of seconds until an operation times out
GetTransferStatus	Return data transfer statistics
IsBlocking	Determine if the client is blocked, waiting for information
IsConnected	Determine if the client is connected to the server
IsInitialized	Determine if the class has been successfully initialized
IsReadable	Determine if data can be read from the server
IsWritable	Determine if data can be written to the server
Login	Login to the server
OpenMessage	Open the specified message for reading on the server
PopEventProc	Callback method that processes events generated by the client
Read	Read data returned by the server
RegisterEvent	Register an event handler for the specified event
Reset	Reset the client and return to a command state
SendMessage	Send a message through the mail server
SetLastError	Set the last error code
SetTimeout	Set the number of seconds until an operation times out
ShowError	Display a message box with a description of the specified error
StoreMessage	Store the contents of a message in the specified file
Write	Write data to the server

CPopClient::CPopClient Method

`CPopClient();`

The **CPopClient** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **PopInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[~CPopClient](#), [IsInitialized](#)

CPopClient::~CPopClient

`~CPopClient();`

The **CPopClient** destructor releases resources allocated by the current instance of the **CPopClient** object. It also uninitialized the library if there are no other concurrent uses of the class.

Remarks

When a **CPopClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cspopv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CPopClient](#)

CPopClient::AttachHandle Method

```
VOID AttachHandle(  
    HCLIENT hClient  
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle already has been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the *hClient* parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[AttachThread](#), [DetachHandle](#), [GetHandle](#)

CPopClient::AttachThread Method

```
DWORD AttachThread(  
    DWORD dwThreadId  
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

See Also

[AttachHandle](#), [Cancel](#), [Connect](#), [DetachHandle](#), [Disconnect](#), [GetHandle](#)

CPopClient::Cancel Method

```
INT Cancel();
```

The **Cancel** method cancels any outstanding blocking operation in the client, causing the blocking method to fail. The application may then retry the operation or terminate the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[IsBlocking](#)

CPOPClient::ChangePassword Method

```
BOOL ChangePassword(  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszOldPassword,  
    LPCTSTR lpszNewPassword  
);
```

The **ChangePassword** method changes the account password for the specified user.

Parameters

lpszUserName

Pointer to a string which specifies the user name of the account who's password will be changed. It is not required that this be the same user name that was used to login to the mail server.

lpszOldPassword

Pointer to a string which specifies the current account password.

lpszNewPassword

Pointer to a string which specifies the new account password. When the method returns, the user's mailbox password will be set to this value.

Return Value

If the method succeeds, it will return a non-zero value. If the method fails, it will return zero. To get extended error information, call **GetLastError**.

Remarks

The **ChangePassword** method is used to change the password associated with the specified account on the server. The method establishes a connection to a separate service running on the server, and does not use the POP3 protocol. For this method to succeed, the server must be configured to allow password changes using the "poppass" service, running on port 106.

Because passwords are sent over the network as clear text, this service is considered to be insecure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [Login](#)

CPOPClient::CloseMessage Method

```
INT CloseMessage(  
    HCLIENT hClient  
);
```

The **CloseMessage** method closes the current message that has been opened or created.

Parameters

hClient

Handle to the client session.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

If an message is being created, this method actually submits the message to the server. Note that the client application is responsible for generating the message headers as well as the body of the message. News messages conform to the same general characteristics of an email message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[OpenMessage](#), [Read](#)

CPopClient::Command Method

```
INT Command(  
    LPCTSTR lpszCommand,  
    LPCTSTR lpszParameter,  
    BOOL bMultiLine  
);
```

The **Command** method sends a command to the server, and returns the result code back to the caller. This method is typically used for site-specific commands not directly supported by the class.

Parameters

lpszCommand

The command which will be executed by the server.

lpszParameter

An optional command parameter. If the command requires more than one parameter, then they should be combined into a single string, with a space separating each parameter. If the command does not accept any parameters, this value may be NULL.

bMultiLine

An optional boolean argument used to specify if multiple lines of data will be returned by the server as the result of the command. Unlike a single line response, which consists of a result code and result string, a multi-line response consists of one or more lines of text, terminated by a special end-of-data marker. If this argument is omitted, *bMultiLine* is FALSE.

Return Value

If the method succeeds, the return value is the result code returned by the server. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

A list of valid commands can be found in the technical specification for the protocol. Many servers will list supported commands when the HELP command is used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetResultCode](#), [GetResultString](#)

CPOPClient::Connect Method

```
BOOL Connect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **Connect** method is used to establish a connection with the server.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 110. For secure connections, the default port number is 995. If the secure port number is specified, an implicit SSL/TLS connection will be established by default.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
POP_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
POP_OPTION_LINEBREAK	Message data that is received from the server is read as individual lines of text terminated by a carriage return and linefeed control sequence. This option can be useful for applications that need to use the lower level network I/O functions and must process the message text on a line-by-line basis. This option is not recommended for most applications because it can have a negative impact on performance when retrieving large messages from the server.
POP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and

	remote port number, default capability selection and how the connection is established.
POP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
POP_OPTION_SECURE	This option specifies that a secure connection should be established with the server and requires that the server support either the SSL or TLS protocol. This option is the same as specifying POP_OPTION_SECURE_EXPLICIT, which initiates the secure session using the STLS command.
POP_OPTION_SECURE_EXPLICIT	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using either the SSL or TLS protocol and the STLS command.
POP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to establish a secure connection with the server. It should only be used when the server expects an implicit SSL connection or does not implement RFC 2595 where the STLS command is used to negotiate a secure connection with the server.
POP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
POP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
POP_OPTION_FREETHREAD	This option specifies that this instance of the class may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the class instance is synchronized across multiple threads.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous socket events that occur. If this argument is NULL, then the client

session will be blocking and no network events will be sent to the client.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the *hEventWnd* argument is NULL, this argument should be specified as WM_NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The use of Windows event notification messages has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

To prevent this method from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **Connect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

If you specify an event notification window, then the client session will be asynchronous. When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
POP_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
POP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
POP_EVENT_READ	Data is available to read by the client. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the calling process is in asynchronous mode.
POP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
POP_EVENT_TIMEOUT	The client has timed out while waiting for a response from the server. Note that under some circumstances this event can be generated for a non-blocking connection, such as when the client is establishing a secure connection.
POP_EVENT_CANCEL	The client has canceled the current operation.

POP_EVENT_COMMAND	The client has processed a command that was sent to the server. The result code and result string can be used to determine if the response to the command. The high word of the IParam parameter should be checked, since this notification message will also be posed if the command cannot be executed.
POP_EVENT_PROGRESS	This event notification is sent periodically during lengthy blocking operations, such as retrieving a complete message from the server.

To cancel asynchronous notification and return the client to a blocking mode, use the **DisableEvents** method.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the class instance is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call methods using that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the POP_OPTION_FREETHREAD option enables any thread to call any method in that instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same instance.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [Disconnect](#), [Login](#)

CPOPClient::CreateSecurityCredentials Method

```
BOOL CreateSecurityCredentials(  
    DWORD dwProtocol,  
    DWORD dwOptions,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName  
);  
  
BOOL CreateSecurityCredentials(  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName  
);  
  
BOOL CreateSecurityCredentials(  
    LPCTSTR lpszCertName  
);
```

The **CreateSecurityCredentials** method establishes the security credentials for the client session.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols.

	This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpzUserName

A pointer to a string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A pointer to a string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A pointer to a string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Example

```
pClient->CreateSecurityCredentials(  
    SECURITY_PROTOCOL_DEFAULT,  
    0,  
    NULL,  
    NULL,  
    lpszCertStore,  
    lpszCertName);
```

```
bConnected = pClient->Connect(lpszHostName,  
                              POP_PORT_SECURE,  
                              POP_TIMEOUT,  
                              POP_OPTION_SECURE);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [DeleteSecurityCredentials](#), [GetSecurityInformation](#), [SECURITYCREDENTIALS](#)

CPopClient::DeleteMessage Method

```
INT DeleteMessage(  
    UINT nMessage  
);
```

The **DeleteMessage** method marks the specified message for deletion from the mailbox.

Parameters

nMessage

Number of message to delete from the server. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method only marks the message for deletion. The message is not actually deleted until the user disconnects from the server. To prevent one or more marked messages from actually being deleted from the mailbox, call the **Reset** method to reset the client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetMessage](#), [GetMessageCount](#), [Reset](#)

CPopClient::DeleteSecurityCredentials Method

```
VOID DeleteSecurityCredentials();
```

The **DeleteSecurityCredentials** method releases the security credentials for the current session.

Parameters

None.

Return Value

None.

Remarks

This method can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated. The security credentials are released when the class destructor is called, so it is normally not required that the application explicitly call this method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateSecurityCredentials](#)

CPopClient::DetachHandle Method

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cspopv10.lib`

See Also

[AttachHandle](#), [GetHandle](#)

CPopClient::DisableEvents Method

```
INT DisableEvents();
```

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the client has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[EnableEvents](#), [RegisterEvent](#)

CPopClient::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[EnableTrace](#)

CPopClient::Disconnect Method

VOID Disconnect();

The **Disconnect** method terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[Connect](#), [IsConnected](#)

CPOPClient::EnableEvents Method

```
INT EnableEvents(  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Constant	Description
POP_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
POP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
POP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
POP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking

	operation. This event is only generated if the client is in asynchronous mode.
POP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
POP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
POP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
POP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

It is not required that the client be placed in asynchronous mode in order to receive command and progress event notifications. To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[DisableEvents](#), [RegisterEvent](#)

CPopClient::EnableTrace Method

```
BOOL EnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **EnableTrace** method enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace method logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableTrace](#)

CPopClient::FreezeEvents Method

```
INT FreezeEvents(  
    BOOL bFreeze  
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the client.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableEvents](#), [EnableEvents](#), [RegisterEvent](#)

CPopClient::GetErrorString Method

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);  
  
INT GetErrorString(  
    DWORD dwErrorCode,  
    CString& strDescription  
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLastError](#), [SetLastError](#), [ShowError](#)

CPopClient::GetHandle Method

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cspopv10.lib`

See Also

[AttachHandle](#), [DetachHandle](#), [IsInitialized](#)

CPopClient::GetHeaderValue Method

```
INT GetHeaderValue(  
    UINT nMessageId,  
    LPCTSTR lpszHeader,  
    LPTSTR lpszValue,  
    INT nMaxLength  
);
```

```
INT GetHeaderValue(  
    UINT nMessageId,  
    LPCTSTR lpszHeader,  
    CString& strValue  
);
```

The **GetHeaderValue** method returns the value of a header field in the specified message.

Parameters

nMessageId

Number of message to retrieve header value from. This value must be greater than zero. The first message in the mailbox is message number one.

lpszHeader

Pointer to a string which specifies the message header to retrieve. The colon should not be included in this string.

lpszValue

Pointer to a string buffer that will contain the value of the specified message header.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the method returns the length of the header field value. If the header field is not present in the message, the method will return a value of zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetHeaderValue** method returns the value of a header field from the specified message. This allows an application to be able to easily determine the value of a header (such as the sender, or the subject of the message) without downloading the entire message header and parsing the contents.

This method uses the XTND XLST command, which is an extension to the POP3 protocol. If this command is not supported by the server, the method will attempt to retrieve the entire message header and return the value for the specified header field. This enables an application to use this method even if the server does not support command extensions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csppv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetMessageHeaders](#), [GetMessageId](#), [GetMessageSender](#)

CPopClient::GetLastError Method

```
DWORD GetLastError();  
  
DWORD GetLastError(  
    CString& strDescription  
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError(0)** when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or POP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

See Also

[GetErrorString](#), [SetLastError](#), [ShowError](#)

CPOPClient::GetMessage Method

```
INT GetMessage(  
    LONG nMessageId,  
    LPBYTE lpBuffer,  
    LPDWORD lpdwLength  
);
```

```
INT GetMessage(  
    LONG nMessageId,  
    HGLOBAL* lpBuffer,  
    LPDWORD lpdwLength  
);
```

```
INT GetMessage(  
    LONG nMessageId,  
    CString& strBuffer  
);
```

The **GetMessage** method retrieves the specified message and copies the contents to a local buffer.

Parameters

nMessageId

Number of message to retrieve from the server. This value must be greater than zero.

lpBuffer

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the method returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpBuffer* parameter. If the *lpBuffer* parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual length of the file that was downloaded.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessage** method is used to retrieve an message from the server and copy it into a local buffer. The method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the message. In this case, the *lpBuffer* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpBuffer* parameter point to a global memory handle which will contain the message data when the method returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. See the example code below.

This method will cause the current thread to block until the complete message has been retrieved,

a timeout occurs or the operation is canceled. During the transfer, the POP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a transfer while it is in progress, use the **GetTransferStatus** method.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Return the message into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter
nResult = pClient->GetMessage(nMessageId, &hgblBuffer, &cbBuffer);

if (nResult != POP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // message text
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

See Also

[EnableEvents](#), [GetMessageHeaders](#), [GetTransferStatus](#), [RegisterEvent](#)

CPOPClient::GetMessageCount Method

```
INT GetMessageCount(  
    UINT *LpnLastMessage,  
    DWORD *LpdwMailboxSize  
);
```

The **GetMessageCount** method returns the number of messages available in the mailbox, the last valid message number in the mailbox and the current size of the mailbox in bytes.

Parameters

lpnLastMessage

Address of a variable that receives the number of the last valid message in the mailbox. If a NULL value is specified, this argument is ignored.

lpdwMailboxSize

Address of a variable that receives the current size of the mailbox. This value will decrease as messages are deleted. If a NULL value is specified, this argument is ignored.

Return Value

If the method succeeds, it returns the number of messages that are currently available. If no messages are available, either because the mailbox is empty or all of the messages have been deleted, this method will return zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[DeleteMessage](#), [GetHeaderValue](#), [GetMessage](#), [GetMessageHeaders](#), [StoreMessage](#)

CPopClient::GetMessageHeaders Method

```
INT GetMessageHeaders(  
    LONG nMessageId,  
    LPBYTE lpHeaders,  
    LPDWORD lpdwLength  
);
```

```
INT GetMessageHeaders(  
    LONG nMessageId,  
    HGLOBAL* lpHeaders,  
    LPDWORD lpdwLength  
);
```

```
INT GetMessageHeaders(  
    LONG nMessageId,  
    CString& strHeaders  
);
```

The **GetMessageHeaders** method retrieves the headers for the specified message from the server.

Parameters

nMessageId

Number of message to retrieve from the server. This value must be greater than zero.

lpHeaders

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the method returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpHeaders* parameter. If the *lpHeaders* parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual length of the message that was downloaded.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessageHeaders** method is used to retrieve an message header block from the server and copy it into a local buffer. The method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the *lpHeaders* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpHeaders* parameter point to a global memory handle which will contain the message headers when the method returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur.

This method will cause the current thread to block until the transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the POP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[GetHeaderValue](#), [GetMessage](#), [GetMessageId](#), [OpenMessage](#)

CPopClient::GetMessageId Method

```
INT GetMessageId(  
    UINT nMessageId,  
    LPTSTR lpszMessageId,  
    INT nMaxLength  
);  
  
INT GetMessageId(  
    UINT nMessageId,  
    CString& strMessageId  
);
```

The **GetMessageId** method returns the message identifier for the specified message.

Parameters

nMessageId

Number of message to retrieve the unique identifier for. This value must be greater than zero. The first message in the mailbox is message number one.

lpszMessageId

Address of a string buffer to receive the message identifier. This should be at least 64 bytes in length. This argument may also be a **CString** object which will contain the message identifier when the method returns.

nMaxLength

The maximum length of the string buffer.

Return Value

If the method succeeds, the return value is the length of the unique identifier string. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessageId** method returns the message identifier from the Message-ID header of the specified message. The returned value is typically a string which specifies the domain, date and timestamp for the message that is created when the message is submitted to the mail server for delivery. To obtain a unique identifier for the message in the mailbox, it is recommended that you use the **GetMessageUid** method instead.

This method uses the XTND XLST command to obtain the value of the "Message-ID" header field. If this command is not supported by the server, the method will attempt to retrieve the entire message header and return the header value. This enables an application to use this method even if the server does not support command extensions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHeaderValue](#), [GetMessage](#), [GetMessageHeaders](#), [GetMessageSender](#), [GetMessageUid](#)

CPopClient::GetMessageSender Method

```
INT GetMessageSender(  
    UINT nMessageId,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

```
INT GetMessageSender(  
    UINT nMessageId,  
    CString& strAddress  
);
```

The **GetMessageSender** method returns the sender's address for the specified message.

Parameters

nMessageId

Number of message to retrieve header value from. This value must be greater than zero. The first message in the mailbox is message number one.

lpszAddress

Pointer to a string buffer that will contain the address of the message sender. This argument may also be a **CString** object which will contain the address when the method returns.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the method returns the length of the address. If the sender cannot be determined, the method will return a value of zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessageSender** method returns the email address of the user who sent the specified message. This allows an application to be able to easily determine the sender, without downloading the entire header block or contents of the message.

This method uses the XSENDER command, which is an extension to the POP3 protocol, to determine the address of the authenticated sender of the message. If the command is not supported, or the server was unable to authenticate the sender, the method will use the XTND XLST command to obtain the value of the "From" header field. If this command is not supported by the server, the method will attempt to retrieve the entire message header and return the value for the specified header field. This enables an application to use this method even if the server does not support command extensions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHeaderValue](#), [GetMessageHeaders](#), [GetMessageId](#), [GetMessageUid](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CPopClient::GetMessageSize Method

```
DWORD GetMessageSize(  
    UINT nMessageId  
);
```

The **GetMessageSize** method returns the size of the specified message.

Parameters

nMessageId

Number of message to retrieve size of. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the method succeeds, the return value is the size of the specified message in bytes. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[GetHeaderValue](#), [GetMessageHeaders](#), [GetMessageId](#), [GetMessageSender](#), [GetTransferStatus](#)

CPopClient::GetMessageUid Method

```
INT GetMessageUid(  
    UINT nMessageId,  
    LPTSTR lpszMessageUID,  
    INT nMaxLength  
);  
  
INT GetMessageUid(  
    UINT nMessageId,  
    CString& strMessageUID  
);
```

The **GetMessageUid** method returns the unique identifier (UID) for the specified message in the current mailbox.

Parameters

nMessageId

Number of message to retrieve the unique identifier for. This value must be greater than zero. The first message in the mailbox is message number one.

lpszMessageUID

Address of a string buffer to receive the unique identifier for the specified message. This should be at least 64 bytes in length. This argument may also be a **CString** object which will contain the UID string when the method returns.

nMaxLength

The maximum length of the string buffer for the message UID.

Return Value

If the method succeeds, it returns a non-zero value. If no unique identifier is assigned to the message, the method will return zero. If an error occurs, the method returns POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetMessageUid** method returns the unique message identifier for the specified message. The returned value is a string which can be used to uniquely identify a specific message in the mailbox across multiple client sessions. This is commonly used by mail clients to determine if they have already retrieved a message from the server in a previous session. The UID can also be used as a key or component of the file name to reference the message after it has been stored on the local system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cspopv10.lib

See Also

[GetHeaderValue](#), [GetMessage](#), [GetMessageHeaders](#), [GetMessageId](#), [GetMessageSender](#)

CPopClient::GetResultCode Method

```
BOOL GetResultCode();
```

The **GetResultCode** method reads the result code returned by the server in response to a command. The result code is a boolean value, and indicates if the operation succeeded or failed.

Parameters

None.

Return Value

If the previous command was successful, the function returns a non-zero value. If the previous command failed, the function returns zero. To get extended error information, call **GetLastError**.

Remarks

Unlike most other Internet application protocols, the Post Office Protocol does not return numeric result codes to indicate success or failure. If a command is successful, the server will respond with the string "+OK" and this is indicated by the **GetResultCode** method returning a non-zero value. If the command fails, the server will respond with the string "-ERR" along with a description of the error, and this is indicated by the method returning a value of zero. The description of the error returned by the server can be obtained by calling the **GetResultString** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[Command](#), [GetResultString](#)

CPopClient::GetResultString Method

```
INT GetResultString(  
    LPTSTR lpszResult,  
    INT cbResult  
);
```

```
INT GetResultString(  
    CString& strResult  
);
```

The **GetResultString** method returns the last message sent by the server along with the result code.

Parameters

lpszResult

A pointer to the buffer that will contain the result string returned by the server. An alternate form of the method accepts a **CString** argument which will contain the result string returned by the server.

cbResult

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetResultString** method is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **GetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Command](#), [GetResultCode](#)

CPopClient::GetSecurityInformation Method

```
BOOL GetSecurityInformation(  
    LPSECURITYINFO LpSecurityInfo  
);
```

The **GetSecurityInformation** method returns security protocol, encryption and certificate information about the current client connection.

Parameters

LpSecurityInfo

A pointer to a [SECURITYINFO](#) structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;  
securityInfo.dwSize = sizeof(SECURITYINFO);  
  
if (pClient->GetSecurityInformation(&securityInfo))  
{  
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)  
    {  
        MessageBox(NULL, _T("The connection is not secure"),  
                    _T("Connection"), MB_OK);  
    }  
    else  
    {  
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)  
        {  
            MessageBox(NULL, _T("The connection is secure"),  
                        _T("Connection"), MB_OK);  
        }  
        else  
        {  
            MessageBox(NULL, _T("The server certificate not valid"),  
                        _T("Connection"), MB_OK);  
        }  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [CreateSecurityCredentials](#), [SECURITYINFO](#)

CPopClient::GetStatus Method

INT GetStatus();

The **GetStatus** method the current status of the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the client status code. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetStatus** method returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Constant	Description
0	POP_STATUS_UNUSED	No connection has been established.
1	POP_STATUS_IDLE	The client is current idle and not sending or receiving data.
2	POP_STATUS_CONNECT	The client is establishing a connection with the server.
3	POP_STATUS_READ	The client is reading data from the server.
4	POP_STATUS_WRITE	The client is writing data to the server.
5	POP_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

See Also

[IsBlocking](#), [IsConnected](#), [IsReadable](#), [IsWritable](#)

CPopClient::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

See Also

[Connect](#), [IsReadable](#), [IsWritable](#), [Read](#), [SetTimeout](#), [Write](#)

CPOPClient::GetTransferStatus Method

```
INT GetTransferStatus(  
    LPPOPTRANSFERSTATUS lpStatus  
);
```

The **GetTransferStatus** method returns information about the current file transfer in progress.

Parameters

lpStatus

A pointer to an [POPTRANSFERSTATUS](#) structure which contains information about the status of the current file transfer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetTransferStatus** method returns information about the current file transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no file currently being transferred, this method will return the status of the last successful transfer made by the client.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[EnableEvents](#), [GetStatus](#), [RegisterEvent](#)

CPopClient::IsBlocking Method

BOOL IsBlocking();

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking method may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Cancel](#), [GetStatus](#), [IsConnected](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#), [Read](#), [Write](#)

CPopClient::IsConnected Method

```
BOOL IsConnected();
```

The **IsConnected** method is used to determine if the client is currently connected to a server.

Parameters

None.

Return Value

If the client is connected to a server, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#)

CPopClient::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[CPopClient](#), [IsBlocking](#), [IsConnected](#)

CPopClient::IsReadable Method

```
BOOL IsReadable(  
    INT nTimeout,  
    LPDWORD lpdwAvail  
);
```

The **IsReadable** method is used to determine if data is available to be read from the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the method returns a non-zero value. If the client cannot read any data, the method returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsConnected](#), [IsInitialized](#), [IsWritable](#), [Write](#)

CPopClient::IsWritable Method

```
BOOL IsWritable(  
    INT nTimeout  
);
```

The **IsWritable** method is used to determine if data can be written to the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the method returns a non-zero value. If the client cannot write any data, the method returns zero.

Remarks

Although this method can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsConnected](#), [IsInitialized](#), [IsReadable](#), [Write](#)

CPopClient::Login Method

```
INT Login(  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    UINT nAuthType  
);
```

The **Login** method authenticates the specified user in on the server. This method must be called after the connection has been established, and before attempting to retrieve messages or perform any other method on the server.

Parameters

lpszUserName

A null terminated string which specifies the user name to be used to authenticate the current client session. For many service providers, the user name is the full email address of the user which owns the mailbox. In some cases, this may only be the portion of their email address before the domain name.

lpszPassword

A null terminated string which specifies the password to be used when authenticating the current client session. If you are using the POP_AUTH_XOAUTH2 or POP_AUTH_BEARER authentication methods, this parameter is not a password, instead it specifies the OAuth 2.0 bearer token provided by the mail service.

nAuthType

Identifies the type of authentication that should be used when the client logs in to the mail server. The following authentication methods are supported:

Constant	Description
POP_AUTH_DEFAULT	The default authentication scheme which sends the username and password as cleartext to the server. Because the user credentials are not encrypted, this method should only be used over a secure connection. This is the same as specifying POP_AUTH_PASS as the authentication method.
POP_AUTH_PASS	The username and password is sent to the server using the USER and PASS commands. This authentication method is supported by most servers and is the default authentication type. The credentials are not encrypted and this method should only be used over secure connections.
POP_AUTH_APOP	The APOP authentication method which uses an MD5 digest of the password. This method has been deprecated is not supported by all servers. It should only be used if required by legacy mail servers which do not support the SASL authentication methods.
POP_AUTH_LOGIN	This authentication type will use the LOGIN method to authenticate the client session. This encodes the username and password in a specific format, but the credentials are not encrypted. It should be used over a secure connection. The server must support the Simple Authentication and Security

	Layer (SASL) mechanism as defined in RFC 4422.
POP_AUTH_PLAIN	This authentication type will use the PLAIN method to authenticate the client session. This encodes the username and password in a specific format, but the credentials are not encrypted. It should be used over a secure connection. The server must support the PLAIN Simple Authentication and Security Layer (SASL) mechanism as defined in RFC 4616.
POP_AUTH_XOAUTH2	This authentication type will use the XOAUTH2 method to authenticate the client session. This authentication method does not require the user password, instead the <i>lpszPassword</i> parameter must specify the bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this method will fail.
POP_AUTH_BEARER	This authentication type will use the OAUTHBEARER method to authenticate the client session as defined in RFC 7628. This authentication method does not require the user password, instead the <i>lpszPassword</i> parameter must specify the bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this method will fail.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The POP_AUTH_LOGIN and POP_AUTH_PLAIN authentication methods require the mail server support the Simple Authentication and Security Layer (SASL) AUTH command as defined in RFC 5034. Most modern mail servers do support one or both of these methods, and they are generally preferred over the POP_AUTH_PASS method when possible. However, for backwards compatibility with legacy servers, the class will default to using POP_AUTH_PASS for client authentication.

You should only use an OAuth 2.0 authentication method if you understand the process of how to request the access token. Obtaining an access token requires registering your application with the mail service provider (e.g.: Microsoft or Google), getting a unique client ID associated with your application and then requesting the access token using the appropriate scope for the service. Obtaining the initial token will typically involve interactive confirmation on the part of the user, requiring they grant permission to your application to access their mail account.

The POP_AUTH_XOAUTH2 and POP_AUTH_BEARER authentication methods are similar, but they are not interchangeable. Both use an OAuth 2.0 bearer token to authenticate the client session, but they differ in how the token is presented to the server. It is currently preferable to use the XOAUTH2 method because it is more widely available and some service providers do not yet support the OAUTHBEARER method.

Your application should not store an OAuth 2.0 bearer token for later use. They have a relatively short lifespan, typically about an hour, and are designed to be used with that session. You should specify offline access as part of the OAuth 2.0 scope if necessary and store the refresh token provided by the service. The refresh token has a much longer validity period and can be used to obtain a new bearer token when needed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [GetMessage](#), [GetMessageCount](#)

CPopClient::OpenMessage Method

```
INT OpenMessage(  
    UINT nMessageId  
);
```

The **OpenMessage** method opens the specified message for reading.

Parameters

nMessageId

Number that specifies which message to open. This value must be greater than zero. The first message in the mailbox is message one.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **OpenMessage** method is used to begin the process of reading the contents of a message from the server. Similar to how a file is opened and read, this method is followed by one or more calls to the **Read** method. When the entire contents of the message has been read, the **CloseMessage** method is used to close the message, completing the transaction on the server.

This is a lower-level method which enables the application to process the message as the contents are being returned by the server. In general, it is recommended that most applications use the **GetMessage** method instead, which provides a simpler interface for retrieving the contents of a message.

It is important to note that you cannot use this method to read the partial contents of a message. Opening a message on the server begins a process where the entire message contents must be read and the message closed before the next command can be issued to the server. If you only want to obtain the headers for a message, use the **GetMessageHeaders** method instead.

Example

```
// Connect to the mail server using the default settings  
if (popClient.Connect(strHostName))  
{  
    // Authenticate the user and display an error if the  
    // server does not accept the username or password  
    if (popClient.Login(strUserName, strPassword) == POP_ERROR)  
    {  
        popClient.ShowError();  
        popClient.Disconnect();  
        return;  
    }  
  
    // Open the specified message  
    if (popClient.OpenMessage(nMessageId) == POP_ERROR)  
    {  
        popClient.ShowError();  
        popClient.Disconnect();  
        return;  
    }  
  
    CString strMessage;
```

```

CString strBuffer;
INT nResult;

// Read the contents of the message in a loop; a return value
// of zero indicates that there is no more data to read
do
{
    if ((nResult = popClient.Read(strBuffer)) > 0)
        strMessage = strMessage + strBuffer;
}
while (nResult > 0);

// If an error occurred while reading reading the message,
// display an error message
if (nResult == POP_ERROR)
{
    popClient.ShowError();
    popClient.Disconnect();
    return;
}

// Close the message and disconnect from the server
popClient.CloseMessage();
popClient.Disconnect();
}

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[CloseMessage](#), [GetMessage](#), [GetMessageHeaders](#), [IsReadable](#), [Read](#)

CPopClient::Read Method

```
INT Read(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Read(  
    CString& strBuffer,  
    INT cbBuffer  
);
```

The **Read** method reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be copied. An alternate form of this method allows a **CString** variable to be passed and data read from the socket will be returned in that string.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When **Read** is called and the client is in non-blocking mode, it is possible that the method will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

See Also

[EnableEvents](#), [IsBlocking](#), [IsReadable](#), [IsWritable](#), [RegisterEvent](#), [Write](#)

CPopClient::RegisterEvent Method

```
INT RegisterEvent(  
    UINT nEventId,  
    POPEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Constant	Description
POP_EVENT_CONNECT	The connection to the server has completed.
POP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
POP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
POP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
POP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
POP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
POP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
POP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more

information about the callback function, see the description of the **PopEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **PopEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This method is typically used to register an event handler that is invoked while a message is being retrieved. The POP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one POP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[DisableEvents](#), [EnableEvents](#), [FreezeEvents](#), [PopEventProc](#)

CPopClient::Reset Method

```
BOOL Reset();
```

The **Reset** method resets the client state and resynchronizes with the server. This method is typically called after an unexpected error has occurred, or an operation has been canceled.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method will prevent any messages marked for deletion from actually being deleted from the mailbox. The client cannot be reset while the client is in a blocked state.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

See Also

[Cancel](#), [DeleteMessage](#), [IsBlocking](#)

CPopClient::SendMessage Method

```
INT SendMessage(  
    LPCTSTR lpszMessage,  
    INT nLength  
);
```

The **SendMessage** method sends a message to the specified recipients.

Parameters

lpszMessage

Pointer to a string buffer which contains the message to be submitted to the mail server for delivery.

nLength

The length of the string buffer. This specifies the number of characters to be written to the mail server. This value must be greater than zero. If this value is -1, then the length of the string up to the terminating null character is used.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SendMessage** method sends a message through the POP3 server using the XTND XMIT command. The specified file must be in the standard format as described in RFC 822, with the recipient addresses specified in the To: and Cc: header fields. Some servers may support blind carbon copies by using addresses specified in a Bcc: header field, and then removing those addresses from the header before delivering the message.

Note that not all POP3 servers support this command, and it is recommended that you use the Simple Mail Transfer Protocol (SMTP) for general mail delivery purposes.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetMessage](#), [StoreMessage](#)

CPopClient::SetLastError Method

```
VOID SetLastError(  
    DWORD dwErrorCode  
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or POP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

See Also

[GetErrorString](#), [GetLastError](#), [ShowError](#)

CPopClient::SetTimeout Method

```
INT SetTimeout(  
    UINT nTimeout  
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[Connect](#), [GetTimeout](#), [IsReadable](#), [IsWritable](#), [Read](#), [Write](#)

CPopClient::ShowError Method

```
INT ShowError(  
    LPCTSTR lpszAppTitle,  
    UINT uType,  
    DWORD dwErrorCode  
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A pointer to a string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetErrorString](#), [GetLastError](#), [SetLastError](#)

CPopClient::StoreMessage Method

```
INT StoreMessage(  
    UINT nMessageId,  
    LPCTSTR lpszFileName  
);
```

The **StoreMessage** method stores a message in the specified file.

Parameters

nMessageId

Number of the message to retrieve. This value must be greater than zero. The first message in the mailbox is message number one.

lpszFileName

Pointer to a string which specifies the file that the message will be stored in. If an empty string or NULL pointer is passed as an argument, the message is copied to the system clipboard.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **StoreMessage** method provides a method of retrieving and storing a message on the local system. The contents of the message is stored as a text file, using the specified file name. This method always causes the caller to block until the entire message has been retrieved, even if the client has been put in asynchronous mode.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetMessage](#), [GetMessageHeaders](#), [GetTransferStatus](#), [SendMessage](#)

CPopClient::Write Method

```
INT Write(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Write(  
    LPCTSTR lpszBuffer  
    INT cbBuffer  
);
```

The **Write** method sends the specified number of bytes to the server.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server. In an alternate form of the method, the pointer is to a string.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero, unless a pointer to a string buffer is passed as the parameter. In that case, if the value is -1, all of the characters in the string, up to but not including the terminating null character, will be sent to the server.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is POP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the next asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableEvents](#), [IsBlocking](#), [IsReadable](#), [IsWritable](#), [Read](#), [RegisterEvent](#)

Post Office Protocol Data Structures

- POPTRANSFERSTATUS
- SECURITYCREDENTIALS
- SECURITYINFO
- SYSTEMTIME

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

POPTRANSFERSTATUS Structure

This structure is used by the [GetTransferStatus](#) method to return information about a message transfer in progress.

```
typedef struct _POPTRANSFERSTATUS
{
    UINT    nMessageId;
    DWORD   dwBytesTotal;
    DWORD   dwBytesCopied;
    DWORD   dwBytesPerSecond;
    DWORD   dwTimeElapsed;
    DWORD   dwTimeEstimated;
} POPTRANSFERSTATUS, *LPPOPTRANSFERSTATUS;
```

Members

nMessageId

The message ID of the current message that is being transferred.

dwBytesTotal

The total number of bytes that will be transferred. If the message is being copied from the server to the local host, this is the size of the message on the server. If the message is being posted to the server, it is the size of message on the local system. If the message size cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the transfer started.

dwTimeEstimated

The estimated number of seconds until the transfer is completed. This is based on the average number of bytes transferred per second.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

SECURITYCREDENTIALS Structure

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on

	what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that

will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A pointer to a null terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A pointer to a null terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A pointer to a null terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A pointer to a null terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
------------	-------------

CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
----	---

MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
----	--

ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.
------	--

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

A pointer to a null terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used for SSH connections

and should always be NULL when establishing a secure connection using SSL or TLS.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* member should be set to CREDENTIAL_STORE_FILENAME and the *lpzCertStore* member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS#12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the *lpzPassword* member or the library will be unable to access the certificate.

Note that the *lpzUserName* and *lpzPassword* members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

The TLS 1.1 and TLS 1.2 protocols are only supported on Windows 7, Windows Server 2008 R2 and later versions of the platform. If these options are specified and the application is running on Windows XP or Windows Vista, the protocol version will be downgraded to TLS 1.0 for backwards compatibility with those platforms.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

SECURITYINFO Structure

This structure contains information about a secure connection that has been established.

```
typedef struct _SECURITYINFO
{
    DWORD        dwSize;
    DWORD        dwProtocol;
    DWORD        dwCipher;
    DWORD        dwCipherStrength;
    DWORD        dwHash;
    DWORD        dwHashStrength;
    DWORD        dwKeyExchange;
    DWORD        dwCertStatus;
    SYSTEMTIME   stCertIssued;
    SYSTEMTIME   stCertExpires;
    LPCTSTR      lpszCertIssuer;
    LPCTSTR      lpszCertSubject;
    LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to **sizeof(SECURITYINFO)** prior to passing the address of this structure to the function. Note that if this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the protocol that was selected to establish the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The

	<p>correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.</p>
SECURITY_PROTOCOL_TLS10	<p>The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS11	<p>The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS12	<p>The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.</p>

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_CIPHER_RC2	The RC2 block cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
SECURITY_CIPHER_DES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher, using 56-bit

	keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher (also known as the Rijndael cipher) is a fixed block size cipher which use a key size of 128, 192 or 256 bits. This cipher is supported on Windows XP SP3 and later versions of the operating system.
SECURITY_CIPHER_SKIPJACK	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
SECURITY_CIPHER_BLOWFISH	The Blowfish block cipher was selected. This is a variable key length cipher up to 448 bits, using a 64-bit block size.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected. One of the following values will be returned:

Constant	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected. One of the following values will be returned:

--	--

Constant	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography. This key exchange algorithm is only supported on Windows XP SP3 and later versions of the operating system.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server. This member only has meaning for connections using the SSL or TLS protocols. One of the following values will be returned:

Constant	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertIssuer

A pointer to a string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertSubject

A pointer to a string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszFingerprint

A pointer to a string which contains a sequence of hexadecimal values that uniquely identify the server. This member is only used when a connection has been established using the Secure Shell (SSH) protocol.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Unicode: Implemented as Unicode and ANSI versions.

SYSTEMTIME Structure

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include windows.h.

Remote Command Protocol Class Library

Execute commands on a server or establish an interactive terminal session.

Reference

- [Class Methods](#)
- [Error Codes](#)

Library Information

Class Name	CRshClient
File Name	CSRSHV10.DLL
Version	10.0.1468.2518
LibID	4893754D-AF7A-4892-9320-D829EEC98DD6
Import Library	CSRSHV10.LIB
Dependencies	None
Standards	RFC 1282

Overview

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

This class library should not be used when connecting to a server over the Internet because the user credentials are sent as unencrypted text. For secure remote command execution and interactive terminal sessions, it is recommended that you use the SocketTools Secure Shell (SSH) class instead.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Remote Command Protocol Class Methods

Class	Description
CRshClient	Constructor which initializes the current instance of the class
~CRshClient	Destructor which releases resources allocated by the class
Method	Description
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified client handle to another thread
Cancel	Cancel the current blocking operation
DetachHandle	Detach the handle for the current instance of this class
DisableEvents	Disable asynchronous event notification
DisableTrace	Disable logging of socket function calls to the trace log
Disconnect	Disconnect from the current server
EnableEvents	Enable asynchronous event notification
EnableTrace	Enable logging of socket function calls to a file
Execute	Execute a command on the server
FreezeEvents	Suspend asynchronous event processing
GetErrorString	Return a description for the specified error code
GetHandle	Return the client handle used by this instance of the class
GetLastError	Return the last error code
GetStatus	Return the current client status
GetTimeout	Return the number of seconds until an operation times out
IsBlocking	Determine if the client is blocked, waiting for information
IsConnected	Determine if the client is connected to the server
IsInitialized	Determine if the class has been successfully initialized
IsReadable	Determine if data can be read from the server
IsWritable	Determine if data can be written to the server
Login	Establish a login session with the specified server
Read	Read data returned by the server
RegisterEvent	Register an event callback function
RshEventProc	Callback method that processes events generated by the client
Search	Search for a specific character sequence in the data stream
SetLastError	Set the last error code
SetTimeout	Set the number of seconds until an operation times out
ShowError	Display a message box with a description of the specified error

Write	Write data to the server
-------	--------------------------

CRshClient::CRshClient

`CRshClient();`

The **CRshClient** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **RshInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[~CRshClient](#), [IsInitialized](#)

CRshClient::~CRshClient

`~CRshClient();`

The **CRshClient** destructor releases resources allocated by the current instance of the **CRshClient** object. It also uninitialized the library if there are no other concurrent uses of the class.

Remarks

When a **CRshClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CRshClient](#)

CRshClient::AttachHandle

```
VOID AttachHandle(  
    HCLIENT hClient  
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle already has been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the *hClient* parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[AttachThread](#), [DetachHandle](#), [GetHandle](#)

CRshClient::AttachThread

```
DWORD AttachThread(  
    DWORD dwThreadId  
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is RSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[AttachHandle](#), [Cancel](#), [DetachHandle](#), [Disconnect](#), [Execute](#), [GetHandle](#), [Login](#)

CRshClient::Cancel

```
INT Cancel();
```

The **Cancel** method cancels any outstanding blocking operation in the client, causing the blocking method to fail. The application may then retry the operation or terminate the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is RSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csrshv10.lib

See Also

[IsBlocking](#)

CRshClient::DetachHandle

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csrshv10.lib`

See Also

[AttachHandle](#), [GetHandle](#)

CRshClient::DisableEvents

```
INT DisableEvents();
```

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is RSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the client has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[EnableEvents](#), [RegisterEvent](#)

CRshClient::DisableTrace

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[EnableTrace](#)

CRshClient::Disconnect

VOID Disconnect();

The **Disconnect** method terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[Execute](#), [Login](#)

CRshClient::EnableEvents

```
INT EnableEvents(  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is RSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Constant	Description
RSH_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
RSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
RSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking

	operation. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
RSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and reconnect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[DisableEvents](#), [RegisterEvent](#)

CRshClient::EnableTrace

```
BOOL EnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **EnableTrace** method enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace method logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableTrace](#)

CRshClient::Execute

```
BOOL Execute(  
    LPCTSTR lpszRemoteHost,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCommand,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

```
BOOL Execute(  
    LPCTSTR lpszRemoteHost,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszCommand,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **Execute** method is used to establish a connection with the server and execute the specified command.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

lpszUserName

A pointer to a string which specifies the username used to authenticate the client session.

lpszPassword

A pointer to a string which specifies the password used to authenticate the client session.

lpszCommand

A pointer to a string which specifies the command to execute on the server.

nRemotePort

The port number the server is listening on. One of the following values should be used:

Constant	Description
RSH_PORT_REXEC	A connection is established with the server using port 512, the rexec service. This service requires that the client provide a username and password to execute the specified command. This is the default port used when calling the version of the method which specifies a password.
RSH_PORT_RSHELL	A connection is established with the server using port 514, the rshell service. This service uses host equivalence to authenticate the user. With host equivalence, the server considers the client to

be equivalent to itself, and as long as the specified user exists on the server, the client is permitted to execute commands on behalf of the user without requiring a password. Host equivalence is configured by the server administrator. This is the default port used when calling the version of the method that does not specify a password.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
RSH_OPTION_RESERVEDPORT	This option specifies that a reserved port should be used to establish the connection. Reserved ports are those port numbers which are less than 1024. This option should be specified when connecting on the RSH_PORT_RSHELL port.
RSH_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
RSH_OPTION_FREETHREAD	This option specifies that this instance of the class may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the class instance is synchronized across multiple threads.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur. If this argument is NULL, then the client session will be blocking and no network events will be sent to the client.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the *hEventWnd* argument is NULL, this argument should be specified as WM_NULL.

Return Value

If the method succeeds, the return value non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The use of Windows event notification messages has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

To prevent this method from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **Execute** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

If you specify an event notification window, then the client session will be asynchronous. When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
RSH_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
RSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
RSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
RSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To cancel asynchronous notification and return the client to a blocking mode, use the **DisableEvents** method.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the class instance is initially attached to the thread

that created it. From that point on, until the it is released, only the owner may call methods using that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the POP_OPTION_FREETHREAD option enables any thread to call any method in that instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same instance.

Example

The following example demonstrates connecting to a server, executing a command on the server, reading the output and storing it in a string. This uses the rexec service on port 512, and requires a username and password.

```
CRshClient rshClient;
BOOL bConnected;

// Establish a connection with the server and execute
// the command as the specified user
bConnected = rshClient.Execute(strHostName,
                              strUserName,
                              strPassword,
                              strCommand);

// If the connection could not be established or the command
// could not be executed, display an error message
if (bConnected == FALSE)
{
    rshClient.ShowError();
    return;
}

CString strBuffer;
CString strResult;
INT nResult;

// Read the output from the command and store it in the strResult
// string; note that a UNIX server terminates each line of output
// with a bare linefeed, which is something that should be kept
// in mind when parsing the string
do
{
    if ((nResult = rshClient.Read(strBuffer)) > 0)
        strResult += strBuffer;
}
while (nResult > 0);

// If there was an error reading the output from the server, then
// display the error and disconnect
if (nResult == RSH_ERROR)
{
    rshClient.ShowError();
    rshClient.Disconnect();
    return;
}
```

```
rshClient.Disconnect();
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Disconnect](#), [Login](#)

CRshClient::FreezeEvents

```
INT FreezeEvents(  
    BOOL bFreeze  
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the client.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is RSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableEvents](#), [EnableEvents](#), [RegisterEvent](#)

CRshClient::GetErrorString

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);  
  
INT GetErrorString(  
    DWORD dwErrorCode,  
    CString& strDescription  
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLastError](#), [SetLastError](#), [ShowError](#)

CRshClient::GetHandle

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csrshv10.lib`

See Also

[AttachHandle](#), [DetachHandle](#), [IsInitialized](#)

CRshClient::GetLastError

```
DWORD GetLastError();  
  
DWORD GetLastError(  
    CString& strDescription  
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError(0)** when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or RSH_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csrshv10.lib

See Also

[GetErrorString](#), [SetLastError](#), [ShowError](#)

CRshClient::GetStatus

INT GetStatus();

The **GetStatus** method the current status of the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the client status code. If the method fails, the return value is RSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetStatus** method returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Constant	Description
0	RSH_STATUS_UNUSED	No connection has been established.
1	RSH_STATUS_IDLE	The client is current idle and not sending or receiving data.
2	RSH_STATUS_CONNECT	The client is establishing a connection with the server.
3	RSH_STATUS_READ	The client is reading data from the server.
4	RSH_STATUS_WRITE	The client is writing data to the server.
5	RSH_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[IsBlocking](#), [IsConnected](#), [IsReadable](#)

CRshClient::GetTimeout

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is RSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[IsReadable](#), [IsWritable](#), [Read](#), [SetTimeout](#), [Write](#)

CRshClient::IsBlocking

BOOL IsBlocking();

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking method may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Cancel](#), [GetStatus](#), [IsConnected](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#), [Read](#), [Write](#)

CRshClient::IsConnected

```
BOOL IsConnected();
```

The **IsConnected** method is used to determine if the client is currently connected to a server.

Parameters

None.

Return Value

If the client is connected to a server, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#)

CRshClient::IsInitialized

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[CRshClient](#), [IsBlocking](#), [IsConnected](#)

CRshClient::IsReadable

```
BOOL IsReadable(  
    INT nTimeout,  
    LPDWORD lpdwAvail  
);
```

The **IsReadable** method is used to determine if data is available to be read from the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the method returns a non-zero value. If the client cannot read any data, the method returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsConnected](#), [IsInitialized](#), [IsWritable](#), [Write](#)

CRshClient::IsWritable

```
BOOL IsWritable(  
    INT nTimeout  
);
```

The **IsWritable** method is used to determine if data can be written to the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the method returns a non-zero value. If the client cannot write any data, the method returns zero.

Remarks

Although this method can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsConnected](#), [IsInitialized](#), [IsReadable](#), [Write](#)

CRshClient::Login

```
BOOL Login(  
    LPCTSTR lpszRemoteHost,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszTerminal,  
    UINT nRemotePort,  
    UINT nTimeout,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **Login** method is used to establish a terminal session with the server.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

lpszUserName

A pointer to a string which specifies the username used to authenticate the client session.

lpszTerminal

A pointer to a string which specifies the terminal type which the client will be identified as using during the session. If no particular terminal emulation is required, this parameter may be NULL.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port of 513 should be used.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur. If this argument is NULL, then the client session will be blocking and no network events will be sent to the client.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the *hEventWnd* argument is NULL, this argument should be specified as WM_NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The use of Windows event notification messages has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

To prevent this method from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **Login** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

The **Login** method uses host equivalence, where the client is permitted to login without requiring a password. Host equivalence must be configured by the server administrator and it is typically restricted to specific users. Note that if configured improperly, host equivalence can introduce a significant security loophole. Refer to your UNIX system documentation for more information about host equivalence and the various remote command services.

On UNIX based systems, the terminal name specified by the *lpszTerminal* parameter corresponds to a termcap or terminfo entry as set in the TERM environment variable. On Windows based systems which implement the rlogin service, this parameter may be ignored and the server will assume that the client is capable of displaying ANSI escape sequences. On VMS systems, the terminal name should correspond to the terminal type used with the SET TERMINAL/DEVICE command.

If this parameter is passed as NULL pointer or an empty string, a default terminal type named "unknown" will be used. On most UNIX and VMS systems this defines a terminal which is not capable of cursor positioning using control or escape sequences. This terminal type may not be recognized and an error may be displayed when the user logs in indicating that the terminal type is invalid.

Refer to the documentation for the server system to determine what terminal type names are available to you. Remember that on UNIX systems, the terminal type is case-sensitive. Some of the more common terminal types are:

Terminal Type	Description
ansi	This terminal type is usually available on UNIX based servers. This specifies that the client is capable of displaying standard ANSI escape sequences for cursor control.
dumb	This terminal type typically specifies a terminal display which does not support control or escape sequences for cursor positioning. If you do not want escape sequences embedded in the data stream and the server returns an error if the terminal type is not specified, try using this terminal type.
pcansi	This terminal type is usually available on UNIX based servers. This specifies that the client is using a PC terminal emulator that supports basic ANSI escape sequences for cursor control. This may also enable escape sequences which can set the display colors.
vt100	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT100. This specifies that the client is capable of emulating a DEC VT100 terminal. The VT100 supports many of the same cursor control sequences as an ANSI terminal.
vt220	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT220. This specifies that the client is capable of emulating a DEC VT220 terminal, which is a later version of the VT100.

vt320	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT320. This specifies that the client is capable of emulating a DEC VT320 terminal, which is similar to the VT100 and VT220 and provides advanced features such as the ability to set display colors.
xterm	This terminal type is may be available on UNIX based servers which have X Windows installed. This specifies that the client is a using the X Windows xterm emulator which supports standard ANSI escape sequences for cursor control.

If you specify an event notification window, then the client session will be asynchronous. When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
RSH_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
RSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
RSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
RSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To cancel asynchronous notification and return the client to a blocking mode, use the **DisableEvents** method.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Disconnect](#), [Execute](#)

CRshClient::Read

```
INT Read(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Read(  
    CString& strBuffer,  
    INT cbBuffer  
);
```

The **Read** method reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be copied. An alternate form of this method allows a **CString** variable to be passed and data read from the socket will be returned in that string.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the method fails, the return value is RSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

When **Read** is called and the client is in non-blocking mode, it is possible that the method will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[EnableEvents](#), [IsBlocking](#), [IsReadable](#), [IsWritable](#), [RegisterEvent](#), [Write](#)

CRshClient::RegisterEvent

```
INT RegisterEvent(  
    UINT nEventId,  
    RSHEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Constant	Description
RSH_EVENT_CONNECT	The connection to the server has completed.
RSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
RSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
RSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **RshEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is

RSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **RshEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a **DWORD_PTR** value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[DisableEvents](#), [EnableEvents](#), [FreezeEvents](#), [RshEventProc](#)

CRshClient::Search

```
BOOL Search(  
    LPCTSTR lpszString  
);  
  
BOOL Search(  
    LPCTSTR lpszString,  
    LPVOID lpBuffer,  
    LPDWORD lpdwLength  
);  
  
BOOL Search(  
    LPCTSTR lpszString,  
    HGLOBAL* lpBuffer,  
    LPDWORD lpdwLength  
);  
  
BOOL Search(  
    LPCTSTR lpszString,  
    CString& strBuffer  
);
```

The **Search** method searches for a specific character sequence in the data stream and stops reading if the sequence is encountered.

Parameters

lpszString

A pointer to a string which specifies the sequence of characters to search for in the data stream. This parameter cannot be NULL or point to an empty string.

lpBuffer

A pointer to a byte buffer which will contain the output from the server, or a pointer to a global memory handle which will reference the output when the method returns. An alternate form of this method accepts a **CString** object which will contain the output when the method returns. If the output from the server is not required, this parameter may be NULL.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpBuffer* parameter. If the *lpBuffer* parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual number of bytes of output stored in the buffer. If the *lpBuffer* parameter is NULL, this parameter should also be NULL.

Return Value

If the method succeeds and the character sequences was found in the data stream, the return value is non-zero. If the method fails or a timeout occurs before the sequence is found, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Search** method searches for a character sequence in the data stream and stops reading when it is found. This is useful when the client wants to automate responses to the server, such as executing a command and processing the output. The method collects the output from the server and stores it in the buffer specified by the *lpBuffer* parameter. When the method returns, the

buffer will contain everything sent by the server up to and including the search string.

The *lpBuffer* parameter may be specified in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the a fixed amount of output. In this case, the *lpBuffer* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer. If the server sends more output than can be stored in the buffer, the remaining output will be discarded.

The second method that can be used is have the *lpBuffer* parameter point to a global memory handle which will contain the output when the method returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. This method is preferred if the client application does not have a general idea of how much output will be generated until the search string is found.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Execute](#), [IsBlocking](#), [IsReadable](#), [Login](#), [Read](#)

CRshClient::SetLastError

```
VOID SetLastError(  
    DWORD dwErrorCode  
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or RSH_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csrshv10.lib

See Also

[GetErrorString](#), [GetLastError](#), [ShowError](#)

CRshClient::SetTimeout

```
INT SetTimeout(  
    UINT nTimeout  
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is RSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[Execute](#), [GetTimeout](#), [IsReadable](#), [IsWritable](#), [Login](#), [Read](#), [Write](#)

CRshClient::ShowError

```
INT ShowError(  
    LPCTSTR lpszAppTitle,  
    UINT uType,  
    DWORD dwErrorCode  
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A pointer to a string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csrshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetErrorString](#), [GetLastError](#), [SetLastError](#)

CRshClient::Write

```
INT Write(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Write(  
    LPCTSTR lpszBuffer  
    INT cbBuffer  
);
```

The **Write** method sends the specified number of bytes to the server.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server. In an alternate form of the method, the pointer is to a string.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero, unless a pointer to a string buffer is passed as the parameter. In that case, if the value is -1, all of the characters in the string, up to but not including the terminating null character, will be sent to the server.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is RSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the next asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableEvents](#), [IsBlocking](#), [IsReadable](#), [IsWritable](#), [Read](#), [RegisterEvent](#)

Secure Shell Protocol Class Library

Establish an interactive terminal session with an SSH server and execute remote commands.

Reference

- [Class Methods](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

Class Name	CSshClient
File Name	CSTSHV10.DLL
Version	10.0.1468.2518
LibID	4C76F231-8ED9-4850-B882-01B77485F43A
Import Library	CSTSHV10.LIB
Dependencies	None
Standards	RFC 4251

Overview

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

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location

on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

SSH Protocol Class Methods

Class	Description
CSshClient	Constructor which initializes the current instance of the class
~CSshClient	Destructor which releases resources allocated by the class
Method	Description
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified client handle to another thread
Break	Send a break signal to the server
Cancel	Cancel the current blocking operation
Connect	Connect to the specified server
Control	Send a control message to the server
CreateSecurityCredentials	Create a new security credentials structure
DeleteSecurityCredentials	Delete a previously created security credentials structure
DetachHandle	Detach the handle for the current instance of this class
DisableEvents	Disable asynchronous event notification
DisableTrace	Disable logging of socket function calls to the trace log
Disconnect	Disconnect from the current server
EnableEvents	Enable asynchronous event notification
EnableTrace	Enable logging of socket function calls to a file
Execute	Execute a command on a server and return the output in the specified buffer
FreezeEvents	Suspend asynchronous event processing
GetErrorString	Return a description for the specified error code
GetExitCode	Return the exit code for the remote session
GetHandle	Return the client handle used by this instance of the class
GetLastError	Return the last error code
GetLineMode	Return the current mode used to send end-of-line character sequences
GetSecurityInformation	Return security information about the current client connection
GetStatus	Return the current client status
GetTimeout	Return the number of seconds until an operation times out
IsBlocking	Determine if the client is blocked, waiting for information
IsConnected	Determine if the client is connected to the server
IsInitialized	Determine if the class has been successfully initialized
IsReadable	Determine if data can be read from the server
IsWritable	Determine if data can be written to the server

Peek	Examine data in the receive buffer, but do not remove it
Read	Read data returned by the server
ReadLine	Read a line of data from the server and return it in a string buffer
RegisterEvent	Register an event callback function
Search	Search for a specific character sequence in the data stream
SetLastError	Set the last error code
SetLineMode	Change how end-of-line character sequences are sent to the server
SetTimeout	Set the number of seconds until an operation times out
ShowError	Display a message box with a description of the specified error
SshEventProc	Callback function that processes events generated by the client
Write	Write data to the server
WriteLine	Write a line of data to the server

CSshClient::CSshClient Method

`CSshClient();`

The **CSshClient** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **SshInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstshv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[~CSshClient](#), [IsInitialized](#)

CSshClient::~CSshClient

`~CSshClient();`

The **CSshClient** destructor releases resources allocated by the current instance of the **CSshClient** object. It also uninitialized the library if there are no other concurrent uses of the class.

Remarks

When a **CSshClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CSshClient](#)

CSshClient::AttachHandle Method

```
VOID AttachHandle(  
    HCLIENT hClient  
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle already has been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the *hClient* parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[AttachThread](#), [DetachHandle](#), [GetHandle](#)

CSshClient::AttachThread Method

```
DWORD AttachThread(  
    DWORD dwThreadId  
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[AttachHandle](#), [Cancel](#), [Connect](#), [DetachHandle](#), [Disconnect](#), [GetHandle](#)

CSshClient::Break Method

```
INT Break();
```

The **Break** method sends a signal to the server.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **Break** method sends a control message to the server which simulates a break signal on a physical terminal. This is used by some operating systems as an instruction to enter a privileged configuration mode. Note that this is not the same as sending an interrupt character such as Ctrl+C to the server. This control code is ignored for SSH 1.0 sessions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Cancel](#), [Control](#), [Read](#), [ReadLine](#), [Write](#), [WriteLine](#)

CSshClient::Cancel Method

```
INT Cancel();
```

The **Cancel** method cancels any outstanding blocking operation in the client, causing the blocking method to fail. The application may then retry the operation or terminate the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[Break](#), [IsBlocking](#)

CSshClient::Connect Method

```
BOOL Connect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSSHOPTIONDATA lpOptions,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **Connect** method establishes a connection with the specified server.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number 22 should be used.

lpszUserName

A pointer to a string which specifies the user name which will be used to authenticate the client session. This parameter must specify a valid user name and cannot be NULL or an empty string.

lpszPassword

A pointer to a string which specifies the password which will be used to authenticate the client session. If the user does not have a password, this parameter can be NULL or an empty string.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SSH_OPTION_NONE	No options specified. A standard terminal session will be established with the default terminal type.
SSH_OPTION_KEEPAIVE	This option specifies the library should attempt to maintain an idle client session for long periods of time. This option is only necessary if you expect that the connection will be held open for more than two hours.
SSH_OPTION_NOPTY	This option specifies that a pseudoterminal (PTY) should not be created for the client session. This option is automatically set if the SSH_OPTION_COMMAND option has been specified.

SSH_OPTION_NOSHELL	This option specifies that a command shell should not be used when executing a command on the server.
SSH_OPTION_NOAUTHRSA	This option specifies that RSA authentication should not be used with SSH-1 connections. This option is ignored with SSH-2 connections and should only be specified if required by the server.
SSH_OPTION_NOPWDNULL	This option specifies the user password cannot be terminated with a null character. This option is ignored with SSH-2 connections and should only be specified if required by the server.
SSH_OPTION_NOREKEY	This option specifies the client should never attempt a repeat key exchange with the server. Some SSH servers do not support rekeying the session, and this can cause the client to become non-responsive or abort the connection after being connected for an hour.
SSH_OPTION_COMPATSID	This compatibility option changes how the session ID is handled during public key authentication with older SSH servers. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions.
SSH_OPTION_COMPATHMAC	This compatibility option changes how the HMAC authentication codes are generated. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions.
SSH_OPTION_TERMINAL	This option specifies the client session will use terminal emulation and the SSHOPTIONDATA structure specifies the characteristics of the virtual terminal. This enables the caller to specify the dimensions of the virtual display (in columns and rows) and the type of terminal that will be emulated. If this option is omitted, the session will default to a virtual display that is 80 columns, 25 rows.
SSH_OPTION_COMMAND	This option specifies the client session will be used to issue a command that is executed on the server, and the output will be returned to the caller. If this option is specified, the session will not be interactive and no pseudoterminal is created for the client. The szCommandLine member of the SSHOPTIONDATA structure specifies the command string that will be sent to the server.
SSH_OPTION_PROXYSERVER	This option specifies the client should establish a connection through a proxy server. The two protocols that are supported are SSH_PROXY_HTTP and SSH_PROXY_TELNET, which specifies the protocol that the proxy connection is created through. The proxy-

	related members of the <code>SSH_OPTIONDATA</code> structure should be set to the appropriate values.
<code>SSH_OPTION_PREFER_IPV6</code>	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
<code>SSH_OPTION_FREETHREAD</code>	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpOptions

A pointer to a `SSH_OPTIONDATA` structure which specifies additional information for one or more options. If no optional data is required, a NULL pointer may be specified.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur. If this argument is NULL, then the client session will be blocking and no network events will be sent to the client.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than `WM_USER` as defined in the Windows header files. If the *hEventWnd* argument is NULL, this argument should be specified as `WM_NULL`.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The use of Windows event notification messages has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

To prevent this method from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **Connect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

If you specify an event notification window, then the client session will be asynchronous. When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
SSH_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
SSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
SSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
SSH_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
SSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
SSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To cancel asynchronous notification and return the client to a blocking mode, use the **DisableEvents** methods.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the class instance is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call methods using that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the SSH_OPTION_FREETHREAD option enables any thread to call any method in that instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same instance.

Example

```
CSshClient sshClient;
SSHOPTIONDATA sshOptions;
BOOL bConnected;
```

```
// Initialize the SSHOPTIONDATA structure and specify the
```

```
// command that should be executed on the server
ZeroMemory(&sshOptions, sizeof(sshOptions));
lstrcpyn(sshOptions.szCommandLine, strCommand, SSH_MAXCOMMANDLEN);

// Establish a connection with the SSH server

bConnected = sshClient.Connect(strHostName,
                               SSH_PORT_DEFAULT,
                               strUserName,
                               strPassword,
                               SSH_TIMEOUT,
                               SSH_OPTION_COMMAND,
                               &sshOptions);

// If the connection attempt fails, then get a description of
// the error and display it in a message box

if (!bConnected)
{
    sshClient.ShowError();
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateSecurityCredentials](#), [Disconnect](#), [GetSecurityInformation](#), [SECURITYCREDENTIALS](#),
[SSHOPTIONDATA](#)

SshClient::Control Method

```
INT Control(  
    DWORD dwControlCode  
);
```

The **SshControl** function sends a control message to the server.

Parameters

dwControlCode

A numeric control code which specifies the control message which should be sent to the server. This may be one of the following values:

Constant	Description
SSH_CONTROL_BREAK	Sends a control message to the server which simulates a break signal on a physical terminal. This is used by some operating systems as an instruction to enter a privileged configuration mode. Note that this is not the same as sending an interrupt character such as Ctrl+C to the server. This control code is ignored for SSH 1.0 sessions.
SSH_CONTROL_NOOP	Sends a control message to the server, but it does not perform any operation. This is typically used by clients to prevent the server from automatically closing a session that has been idle for a long period of time.
SSH_CONTROL_EOF	Sends a control message to the server indicating that the client has finished sending data. Note that this option is normally not used with interactive terminal sessions, and should only be used when required by the server.
SSH_CONTROL_PING	Sends a control message to the server which is used to test whether or not the server is responsive to the client. This is typically used by clients to attempt to detect if the connection to the server is still active.
SSH_CONTROL_REKEY	Sends a control message to the server requesting that the key exchange be performed again. This control code is ignored for SSH 1.0 sessions.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SSH_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

The **Control** method enables an application to send control messages to the server, which can cause it to take specific actions such as simulate a terminal break or request that the key exchanged be performed again. Some control messages are not supported by the SSH 1.0 protocol, in which case the control message is ignored.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[Break](#), [Cancel](#), [IsBlocking](#)

CSshClient::CreateSecurityCredentials Method

```
BOOL CreateSecurityCredentials(  
    DWORD dwProtocol,  
    DWORD dwOptions,  
    LPCTSTR LpszKeyFile,  
    LPCTSTR LpszPassword  
);
```

```
BOOL CreateSecurityCredentials(  
    LPCTSTR LpszKeyFile,  
    LPCTSTR LpszPassword  
);
```

```
BOOL CreateSecurityCredentials(  
    LPCTSTR LpszKeyFile  
);
```

The **CreateSecurityCredentials** method establishes the security credentials for the client session.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_SSH	Select either version 1.0 or 2.0 of the Secure Shell protocol. The actual protocol version that is selected is determined automatically. This is the recommended value.
SECURITY_PROTOCOL_SSH1	Version 1.0 of the Secure Shell protocol. This protocol has been deprecated and its use widely used. It is not recommended that this protocol be used when establishing secure connections.
SECURITY_PROTOCOL_SSH2	Version 2.0 of the Secure Shell protocol. This is currently the most commonly used version of the protocol, and most servers will require this version when establishing a connection.

dwOptions

Credentials options. This argument is reserved for future use. Set it to zero when using this method.

lpszKeyFile

A pointer to a string which specifies the name of a private key file that used when authenticating the client connection. If a private key is not required, value of NULL should be specified.

lpszPassword

A pointer to a string which specifies the password for the private key file. A value of NULL specifies that no password is required.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Example

```
pClient->CreateSecurityCredentials(  
    SECURITY_PROTOCOL_SSH,  
    0,  
    lpszKeyFile,  
    lpszPassword);  
  
bConnected = pClient->Connect(lpszHostName,  
    SSH_PORT_DEFAULT,  
    lpszUserName,  
    lpszPassword,  
    SSH_TIMEOUT,  
    SSH_OPTION_KEEPAIVE);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [DeleteSecurityCredentials](#), [GetSecurityInformation](#), [SECURITYCREDENTIALS](#)

CSshClient::DeleteSecurityCredentials Method

```
VOID DeleteSecurityCredentials();
```

The **DeleteSecurityCredentials** method releases the security credentials for the current session.

Parameters

None.

Return Value

None.

Remarks

This method can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated. The security credentials are released when the class destructor is called, so it is normally not required that the application explicitly call this method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateSecurityCredentials](#)

CSshClient::DetachHandle Method

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstshv10.lib`

See Also

[AttachHandle](#), [GetHandle](#)

CSshClient::DisableEvents Method

```
INT DisableEvents();
```

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the client has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[EnableEvents](#), [RegisterEvent](#)

CSshClient::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[EnableTrace](#)

CSshClient::Disconnect Method

VOID Disconnect();

The **Disconnect** method terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[Connect](#), [IsConnected](#)

CSshClient::EnableEvents Method

```
INT EnableEvents(  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Constant	Description
SSH_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
SSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
SSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
SSH_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking

	operation. This event is only generated if the client is in asynchronous mode.
SSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
SSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and reconnect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[DisableEvents](#), [RegisterEvent](#)

CSshClient::EnableTrace Method

```
BOOL EnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **EnableTrace** method enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace method logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableTrace](#)

CSshClient::Execute Method

```
INT Execute(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCommandLine,  
    UINT nTimeout,  
    DWORD dwOptions,  
    HGLOBAL* lphgblBuffer,  
    LPDWORD lpdwLength  
);
```

```
INT Execute(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCommandLine,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPBYTE lpBuffer,  
    LPDWORD lpdwLength  
);
```

```
INT Execute(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCommandLine,  
    UINT nTimeout,  
    DWORD dwOptions,  
    CString& strBuffer  
);
```

The **Execute** method executes a command on the server and returns the output in the specified buffer.

Parameters

lpszRemoteHost

A pointer to a string which specifies the name of the server. This may either be a fully-qualified domain name, or an IP address. This parameter cannot be NULL.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number 22 should be used.

lpszUserName

A pointer to a string which specifies the user name which will be used to authenticate the client session.

lpszPassword

A pointer to a string which specifies the password which will be used to authenticate the client

session.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SSH_OPTION_NONE	No options specified. A standard terminal session will be established with the default terminal type.
SSH_OPTION_KEEPAIVE	This option specifies the library should attempt to maintain an idle client session for long periods of time. This option is only necessary if you expect that the connection will be held open for more than two hours.
SSH_OPTION_NOPTY	This option specifies that a pseudoterminal (PTY) should not be created for the client session. This option is automatically set if the SSH_OPTION_COMMAND option has been specified.
SSH_OPTION_NOSHELL	This option specifies that a command shell should not be used when executing a command on the server.
SSH_OPTION_NOAUTHRSA	This option specifies that RSA authentication should not be used with SSH-1 connections. This option is ignored with SSH-2 connections and should only be specified if required by the server.
SSH_OPTION_NOPWDNULL	This option specifies the user password cannot be terminated with a null character. This option is ignored with SSH-2 connections and should only be specified if required by the server.
SSH_OPTION_NOREKEY	This option specifies the client should never attempt a repeat key exchange with the server. Some SSH servers do not support rekeying the session, and this can cause the client to become non-responsive or abort the connection after being connected for an hour.
SSH_OPTION_COMPATSID	This compatibility option changes how the session ID is handled during public key authentication with older SSH servers. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions.
SSH_OPTION_COMPATHMAC	This compatibility option changes how the HMAC authentication codes are generated. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions.

lpBuffer, *lphgblBuffer* or *strBuffer*

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns. If the application is using MFC, then a CString variable may also be specified, in which case the data is returned in the string

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvBuffer* parameter. If the *lpvBuffer* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the file that was downloaded.

lpCredentials

A pointer to a [SECURITYCREDENTIALS](#) structure which specifies additional security-related information required to establish the connection. This parameter may be NULL, in which case default values will be used. Note that the *dwSize* member must be initialized to the size of the **SECURITYCREDENTIALS** structure that is being passed to the function.

Return Value

If the function succeeds, the return value is the exit code from the program that was executed on the server. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

The **Execute** method is used to execute a command on a server, read the output from that command and copy it into a local buffer. This method cannot be used if the connection to the server must be established through a proxy server; if a proxy server must be used, then you should use the **Connect** method to establish the connection, and then use either the **Read** or **ReadLine** methods to read the output.

This method may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the command output. In this case, the *lpvBuffer* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lphgblBuffer* parameter point to a global memory handle which will contain the output when the function returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

When the command output is being read from the server, this method will automatically convert the data to match the end-of-line convention used on the Windows platform. This is useful when executing a command on a UNIX based system where the end-of-line is indicated by a single linefeed, while on Windows it is a carriage-return and linefeed pair. If the output contains embedded nulls or escape sequences, then this conversion will not be performed.

This method will cause the current thread to block until the command completes or a timeout occurs.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;  
LPBYTE lpBuffer = (LPBYTE)NULL;  
DWORD cbBuffer = 0;
```

```
// Execute a command on the server and return the data into block
// of global memory allocated by the GlobalAlloc function; the handle
// to this memory will be returned in the hgblBuffer parameter
nResult = sshClient.Execute(strHostName,
                            SSH_PORT_DEFAULT,
                            strUserName,
                            strPassword,
                            strCommandLine,
                            SSH_TIMEOUT,
                            SSH_OPTION_NONE,
                            &hgblBuffer,
                            &cbBuffer);

if (nResult != SSH_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // resource data
    lpBuffer = (LPBYTE)::GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    ::GlobalUnlock(hgblBuffer);
    ::GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Server: Requires Windows Server 2003 or Windows 2000 Server.

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [GetExitCode](#), [Read](#), [ReadLine](#), [Write](#), [WriteLine](#), [SECURITYCREDENTIALS](#), [SSHOPTIONDATA](#)

CSshClient::FreezeEvents Method

```
INT FreezeEvents(  
    BOOL bFreeze  
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the client.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableEvents](#), [EnableEvents](#), [RegisterEvent](#)

CSshClient::GetErrorString Method

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);  
  
INT GetErrorString(  
    DWORD dwErrorCode,  
    CString& strDescription  
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLastError](#), [SetLastError](#), [ShowError](#)

CSshClient::GetExitCode Method

```
INT GetExitCode();
```

The **GetExitCode** function returns the exit code for the remote session.

Parameters

None.

Return Value

If the method succeeds, the return value is the numeric exit code. If the function fails, the return value is SSH_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

This method should only be called after the command has completed and the **Read** method has returned a value of zero. In most cases, an exit code value of zero indicates success, while any other value indicates an error condition.

Note that the actual value is application dependent and is only meaningful in the context of that particular program. A program may choose or use exit codes in a non-standard way, such as having certain non-zero values indicate success.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[GetStatus](#)

CSshClient::GetHandle Method

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstshv10.lib`

See Also

[AttachHandle](#), [DetachHandle](#), [IsInitialized](#)

CSshClient::GetLastError Method

```
DWORD GetLastError();  
  
DWORD GetLastError(  
    CString& strDescription  
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError(0)** when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or SSH_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[GetErrorString](#), [SetLastError](#), [ShowError](#)

CSshClient::GetLineMode Method

```
INT GetLineMode();
```

The **GetLineMode** method returns the current line mode.

Parameters

None.

Return Value

If the method succeeds, the return value is the current line mode. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetLineMode** method returns an integer value that specifies how end-of-line character sequences are sent to the server. For more information about how newlines are processed by the class and the available options, refer to the **SetLineMode** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[SetLineMode](#), [Write](#), [WriteLine](#)

CSshClient::GetSecurityInformation Method

```
BOOL GetSecurityInformation(  
    LPSECURITYINFO LpSecurityInfo  
);
```

The **GetSecurityInformation** method returns security protocol, encryption and certificate information about the current client connection.

Parameters

LpSecurityInfo

A pointer to a [SECURITYINFO](#) structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value `SECURITY_PROTOCOL_NONE`.

Example

The following example demonstrates how to obtain the fingerprint for the server:

```
SECURITYINFO securityInfo;  
  
ZeroMemory(&securityInfo, sizeof(SECURITYINFO));  
securityInfo.dwSize = sizeof(SECURITYINFO);  
  
if (sshClient.GetSecurityInformation(&securityInfo))  
{  
    if (securityInfo.lpszFingerprint != NULL)  
    {  
        TCHAR szMessage[256];  
        wsprintf(szMessage, _T("The fingerprint is %s",  
securityInfo.lpszFingerprint);  
        MessageBox(NULL, szMessage, "Connection", MB_OK);  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [CreateSecurityCredentials](#), [SECURITYINFO](#)

CSshClient::GetStatus Method

INT GetStatus();

The **GetStatus** method the current status of the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the client status code. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetStatus** method returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Constant	Description
0	SSH_STATUS_UNUSED	No connection has been established.
1	SSH_STATUS_IDLE	The client is current idle and not sending or receiving data.
2	SSH_STATUS_CONNECT	The client is establishing a connection with the server.
3	SSH_STATUS_AUTHENTICATE	The client is authenticating the session with the server.
4	SSH_STATUS_READ	The client is reading data from the server.
5	SSH_STATUS_WRITE	The client is writing data to the server.
6	SSH_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[IsBlocking](#), [IsConnected](#), [IsReadable](#), [IsWritable](#)

CSshClient::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[Connect](#), [IsReadable](#), [IsWritable](#), [Read](#), [SetTimeout](#), [Write](#)

CSshClient::IsBlocking Method

BOOL IsBlocking();

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking method may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Cancel](#), [GetStatus](#), [IsConnected](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#), [Read](#), [Write](#)

CSshClient::IsConnected Method

```
BOOL IsConnected();
```

The **IsConnected** method is used to determine if the client is currently connected to a server.

Parameters

None.

Return Value

If the client is connected to a server, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#)

CSshClient::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[CSshClient](#), [IsBlocking](#), [IsConnected](#)

CSshClient::IsReadable Method

```
BOOL IsReadable(  
    INT nTimeout,  
    LPDWORD lpdwAvail  
);
```

The **IsReadable** method is used to determine if data is available to be read from the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the method returns a non-zero value. If the client cannot read any data, the method returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 8192 bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsConnected](#), [IsInitialized](#), [IsWritable](#), [Write](#)

CSshClient::IsWritable Method

```
BOOL IsWritable(  
    INT nTimeout  
);
```

The **IsWritable** method is used to determine if data can be written to the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the method returns a non-zero value. If the client cannot write any data, the method returns zero.

Remarks

Although this method can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsConnected](#), [IsInitialized](#), [IsReadable](#), [Write](#)

CSshClient::Peek Method

```
INT Peek(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Peek(  
    CString& strBuffer,  
    INT cbBuffer  
);
```

The **Peek** method reads the specified number of bytes from the server and copies them into the buffer, but it does not remove the data from the internal receive buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be copied. An alternate form of this method allows a **CString** variable to be passed and data read from the socket will be returned in that string.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the method fails, the return value is `SSH_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **Peek** method can be used to examine the data that is available to be read from the internal receive buffer. If there is no data in the receive buffer at that time, a value of zero is returned. It should be noted that this differs from the **Read** method, where a return value of zero indicates that there is no more data available to be read and the connection has been closed. The **Peek** method will never cause the client to block, and so may be safely used with asynchronous connections.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstshv10.lib`

See Also

[IsBlocking](#), [IsReadable](#), [Read](#), [ReadLine](#), [Write](#), [WriteLine](#)

CSshClient::Read Method

```
INT Read(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Read(  
    CString& strBuffer,  
    INT cbBuffer  
);
```

The **Read** method reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be copied. An alternate form of this method allows a **CString** variable to be passed and data read from the socket will be returned in that string.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

When **Read** is called and the client is in non-blocking mode, it is possible that the method will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[IsBlocking](#), [IsReadable](#), [Peek](#), [ReadLine](#), [Write](#), [WriteLine](#)

CSshClient::ReadLine Method

```
BOOL ReadLine(  
    LPTSTR lpszBuffer,  
    LPINT lpnLength  
);  
  
BOOL ReadLine(  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);  
  
BOOL ReadLine(  
    CString& strBuffer,  
    INT nMaxLength  
);
```

The **ReadLine** method reads up to a line of data from the server and returns it in a string buffer.

Parameters

lpszBuffer

Pointer to the string buffer that will contain the data when the method returns. The string will be terminated with a null character, and will not contain the end-of-line characters. An alternate form of the method accepts a **CString** argument which will contain the line of text returned by the server.

lpnLength

A pointer to an integer value which specifies the length of the buffer. The value should be initialized to the maximum number of characters that can be copied into the string buffer, including the terminating null character. When the method returns, its value will updated with the actual length of the string.

nMaxLength

An integer value which specifies the maximum length of the buffer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ReadLine** method reads data from the server and copies into a specified string buffer. Unlike the **Read** method which reads arbitrary bytes of data, this method is specifically designed to return a single line of text data in a string. When an end-of-line character sequence is encountered, the method will stop and return the data up to that point. The string buffer is guaranteed to be null-terminated and will not contain the end-of-line characters.

There are some limitations when using **ReadLine**. The method should only be used to read text, never binary data. In particular, the method will discard nulls, linefeed and carriage return control characters. The Unicode version of this method will return a Unicode string, however it does not support reading raw Unicode data from the socket. Any data read from the socket is internally buffered as octets (eight-bit bytes) and converted to Unicode using the **MultiByteToWideChar** function.

This method will force the thread to block until an end-of-line character sequence is processed, the read operation times out or the server closes its end of the socket connection. If this method is

called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, read the data and then restore the socket to asynchronous operation. If another socket operation is attempted while **ReadLine** is blocked waiting for data from the server, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **Read** and **ReadLine** method calls can be intermixed, however be aware that **Read** will consume any data that has already been buffered by the **ReadLine** method and this may have unexpected results.

Unlike the **Read** method, it is possible for data to be returned in the string buffer even if the return value is zero. Applications should check the length of the string to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the return value.

Example

```
CString strBuffer;
BOOL bResult;

do
{
    bResult = pClient->ReadLine(strBuffer);

    if (strBuffer.GetLength() > 0)
    {
        // Process the line of data returned in the string
        // buffer; the string is always null-terminated
    }
} while (bResult);

DWORD dwError = pClient->GetLastError();

if (dwError == ST_ERROR_CONNECTION_CLOSED)
{
    // The server has closed its side of the connection and
    // there is no more data available to be read
}
else if (dwError != 0)
{
    // An error has occurred while reading a line of data
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IsReadable](#), [Peek](#), [Read](#), [Write](#), [WriteLine](#)

CSshClient::RegisterEvent Method

```
INT RegisterEvent(  
    UINT nEventId,  
    SSHEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Constant	Description
SSH_EVENT_CONNECT	The connection to the server has completed.
SSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
SSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
SSH_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
SSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
SSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **SshEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is

SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **SshEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[DisableEvents](#), [EnableEvents](#), [FreezeEvents](#), [SshEventProc](#)

CSshClient::Search Method

```
BOOL Search(  
    LPCTSTR lpszString  
);
```

```
BOOL Search(  
    LPCTSTR lpszString,  
    LPBYTE lpBuffer,  
    LPDWORD lpdwLength  
);
```

```
BOOL Search(  
    LPCTSTR lpszString,  
    HGLOBAL* lpBuffer,  
    LPDWORD lpdwLength  
);
```

```
BOOL Search(  
    LPCTSTR lpszString,  
    CString& strBuffer  
);
```

The **Search** method searches for a specific character sequence in the data stream and stops reading if the sequence is encountered.

Parameters

lpszString

A pointer to a string which specifies the sequence of characters to search for in the data stream. This parameter cannot be NULL or point to an empty string.

lpBuffer

A pointer to a byte buffer which will contain the output from the server, or a pointer to a global memory handle which will reference the output when the method returns. If the output from the server is not required, this parameter may be NULL.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpBuffer* parameter. If the *lpBuffer* parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual number of bytes of output stored in the buffer. If the *lpBuffer* parameter is NULL, this parameter should also be NULL.

Return Value

If the method succeeds and the character sequences was found in the data stream, the return value is non-zero. If the method fails or a timeout occurs before the sequence is found, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Search** method searches for a character sequence in the data stream and stops reading when it is found. This is useful when the client wants to automate responses to the server, such as logging in a user and executing a command. The method collects the output from the server and stores it in the buffer specified by the *lpBuffer* parameter. When the method returns, the buffer will contain everything sent by the server up to and including the search string.

The *lpBuffer* parameter may be specified in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the a fixed amount of output. In this case, the *lpBuffer* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer. If the server sends more output than can be stored in the buffer, the remaining output will be discarded.

The second method that can be used is have the *lpBuffer* parameter point to a global memory handle which will contain the output when the method returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. This method is preferred if the client application does not have a general idea of how much output will be generated until the search string is found.

Example

```
LPCTSTR lpszCommand = _T("/bin/ls -l\r\n");
LPCTSTR lpszPrompt = _T("$ ");
HGLOBAL hglbOutput = NULL;
DWORD cbOutput = 0;
BOOL bResult;

// Search for a command prompt issued by the server

bResult = pClient->Search(lpszPrompt, NULL, NULL, 0);

// If the shell prompt was found, issue the command
// and capture the output into the hglbBuffer global
// memory buffer; the cbBuffer variable will contain
// the actual number of bytes in the buffer when the
// function returns

if (bResult)
{
    pClient->Write((LPBYTE)lpszCommand, lstrlen(lpszCommand));

    bResult = pClient->Search(lpszPrompt,
                             &hglbOutput,
                             &cbOutput,
                             0);
}

// Write the contents of the output buffer to the
// standard output stream

if (bResult)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hglbBuffer);

    if (lpBuffer)
        fwrite(lpBuffer, 1, cbBuffer, stdout);

    GlobalUnlock(hglbBuffer);
    GlobalFree(hglbBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IsBlocking](#), [IsReadable](#), [Peek](#), [Read](#), [ReadLine](#), [Write](#), [WriteLine](#)

CSshClient::SetLastError Method

```
VOID SetLastError(  
    DWORD dwErrorCode  
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or SSH_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstshv10.lib

See Also

[GetErrorString](#), [GetLastError](#), [ShowError](#)

CSshClient::SetLineMode Method

```
INT SetLineMode(  
    INT nLineMode  
);
```

The **SetLineMode** method changes the current line mode for the client session.

Parameters

nLineMode

An integer value which specifies how the newlines are sent by the class. It must be one of the following values:

Value	Constant	Description
0	SSH_NEWLINE_DEFAULT	There are no changes to how data is sent to the server. Any carriage return or linefeed characters that are sent using the Write method will be sent as-is. The WriteLine method will terminate each line of text with a carriage return and linefeed (CRLF) sequence. This is the default line mode that is set when a new connection is established.
1	SSH_NEWLINE_CR	A carriage return is used as the end-of-line character. Any data sent using the Write method that contains only a linefeed (LF) character or a carriage return and linefeed (CRLF) sequence to indicate the end-of-line will be replaced by a carriage return (CR) character. The WriteLine method will terminate each line of text with a single carriage return character.
2	SSH_NEWLINE_LF	A linefeed is used as the end-of-line character. Any data sent using the Write method that contains only a carriage return (CR) character or a carriage return and linefeed (CRLF) sequence to indicate the end-of-line will be replaced by a linefeed (LF) character. The WriteLine method will terminate each line of text with a single linefeed character.
3	SSH_NEWLINE_CRLF	A carriage return and linefeed (CRLF) character sequence is used to indicate the end-of-line. Any data sent using the Write method that contains only a carriage return (CR) or linefeed (LF) will be replaced by a carriage return and linefeed. The WriteLine method will terminate each line of text with a carriage return and linefeed sequence.

Return Value

If the method succeeds, the return value is the previous line mode for the client session. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a connection is initially established with the server, it determines what characters are used to indicate the end-of-line and how they are displayed. On UNIX based systems, this is controlled by the settings for the pseudo-terminal that is allocated for the client session, and can be changed using the **stty** command. In most cases, the client line mode can be left at the default. However, in some cases you may need to change the line mode, particularly if you intend to send data from a Windows text file or copied from the clipboard.

Windows uses a carriage return and linefeed (CRLF) sequence to indicate the end-of-line and a UNIX based server may interpret that as multiple newlines. To prevent this, use the **SetLineMode** method to change the current line mode to **SSH_NEWLINE_CR** and the CRLF sequence in the text will be replaced by a single carriage return.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[GetLineMode](#), [Write](#), [WriteLine](#)

CSshClient::SetTimeout Method

```
INT SetTimeout(  
    UINT nTimeout  
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `SSH_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstshv10.lib`

See Also

[Connect](#), [GetTimeout](#), [IsReadable](#), [IsWritable](#), [Read](#), [Write](#)

CSshClient::ShowError Method

```
INT ShowError(  
    LPCTSTR lpszAppTitle,  
    UINT uType,  
    DWORD dwErrorCode  
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A pointer to a string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetErrorString](#), [GetLastError](#), [SetLastError](#)

CSshClient::Write Method

```
INT Write(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Write(  
    LPCTSTR lpszBuffer  
    INT cbBuffer  
);
```

The **Write** method sends the specified number of bytes to the server.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server. In an alternate form of the method, the pointer is to a string.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero, unless a pointer to a string buffer is passed as the parameter. In that case, if the value is -1, all of the characters in the string, up to but not including the terminating null character, will be sent to the server.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is SSH_ERROR. To get extended error information, call **GetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the next asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IsBlocking](#), [IsWritable](#), [Peek](#), [Read](#), [ReadLine](#), [SetLineMode](#), [WriteLine](#)

CSshClient::WriteLine Method

```
BOOL WriteLine(  
    LPCTSTR lpszBuffer,  
    LPINT lpnLength  
);
```

The **WriteLine** function sends a line of text to the server, terminated by a carriage-return and linefeed.

Parameters

lpszBuffer

The pointer to a string buffer which contains the data that will be sent to the server. All characters up to, but not including, the terminating null character will be written to the socket. The data will always be terminated with a carriage-return and linefeed control character sequence. If this parameter points to an empty string or NULL pointer, then a only a carriage-return and linefeed are written to the socket.

lpnLength

A pointer to an integer value which will contain the number of characters written to the socket, including the carriage-return and linefeed sequence. If this information is not required, a NULL pointer may be specified.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **WriteLine** method writes a line of text to the server and terminates the line with a carriage-return and linefeed control character sequence. Unlike the **Write** method which writes arbitrary bytes of data to the socket, this method is specifically designed to write a single line of text data from a string.

If the *lpszBuffer* string is terminated with a linefeed (LF) or carriage return (CR) character, it will be automatically converted to a standard CRLF end-of-line sequence. Because the string will be sent with a terminating CRLF sequence, the value returned in the *lpnLength* parameter will typically be larger than the original string length (reflecting the additional CR and LF characters), unless the string was already terminated with CRLF.

There are some limitations when using **WriteLine**. This method should only be used to send text, never binary data. In particular, it will discard nulls and append linefeed and carriage return control characters to the data stream. The Unicode version of this method will accept a Unicode string, however it does not support writing raw Unicode data to the socket. Unicode strings will be automatically converted to UTF-8 encoding using the **WideCharToMultiByte** function and then written to the socket as a stream of bytes.

This method will force the thread to block until the complete line of text has been written, the write operation times out or the server aborts the connection. If this method is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, send the data and then restore the socket to asynchronous operation. If another socket operation is attempted while **WriteLine** is blocked sending data to the server, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker

threads to manage each connection.

The **Write** and **WriteLine** methods can be safely intermixed.

Unlike the **Write** function, it is possible for data to have been written to the socket if the return value is zero. For example, if a timeout occurs while the method is waiting to send more data to the server, it will return zero; however, some data may have already been written prior to the error condition. If this is the case, the *lpnLength* argument will specify the number of characters actually written up to that point.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IsWritable](#), [Peek](#), [Read](#), [ReadLine](#), [SetLineMode](#), [Write](#)

SSH Protocol Data Structures

- SECURITYCREDENTIALS
- SECURITYINFO
- SSOPTIONDATA

SECURITYCREDENTIALS Structure

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A value which specifies which security protocols are supported:

Constant	Description
SECURITY_PROTOCOL_SSH	Either version 1.0 or 2.0 of the Secure Shell protocol should be used when establishing the connection. The correct protocol is automatically selected based on the version of the protocol that is supported by the server.
SECURITY_PROTOCOL_SSH1	The Secure Shell 1.0 protocol should be used when establishing the connection. This is an older version of the protocol which should not be used unless explicitly required by the server. Most modern SSH server support version 2.0 of the protocol.
SECURITY_PROTOCOL_SSH2	The Secure Shell 2.0 protocol should be used when establishing the connection. This is the default version of the protocol that is supported by most SSH servers.

dwOptions

This structure member is reserved for use with SSL and TLS connections and should always be initialized to zero.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A pointer to a null terminated string which specifies the hostname that will be used when

validating a server certificate. This member should always be initialized as a NULL pointer for connections using the SSH protocol.

lpszUserName

A pointer to a null terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A pointer to a null terminated string which specifies the password needed to access the certificate. Currently this member is only used if a private key file has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A pointer to a null terminated string which specifies the name of the certificate store to open. This member should always be initialized as a NULL pointer for connections using the SSH protocol.

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. This member should always be initialized as a NULL pointer for connections using the SSH protocol.

lpszKeyFile

A pointer to a null terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used with the SSH protocol. If the member is NULL, then no private key is used.

Remarks

A client application typically only needs to create this structure if the server requires that the client provide a private key as part of the process of negotiating the secure session.

Note that the *lpszUserName* and *lpszPassword* members are values which are used to access the private key file. They are not the credentials which are used when establishing the connection with the server or authenticating the client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

SECURITYINFO Structure

This structure contains information about a secure connection that has been established with a server.

```
typedef struct _SECURITYINFO
{
    DWORD        dwSize;
    DWORD        dwProtocol;
    DWORD        dwCipher;
    DWORD        dwCipherStrength;
    DWORD        dwHash;
    DWORD        dwHashStrength;
    DWORD        dwKeyExchange;
    DWORD        dwCertStatus;
    SYSTEMTIME   stCertIssued;
    SYSTEMTIME   stCertExpires;
    LPCTSTR      lpszCertIssuer;
    LPCTSTR      lpszCertSubject;
    LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to **sizeof(SECURITYINFO)** prior to passing the address of this structure to the function. Note that if this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the protocol that was selected to establish the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_PROTOCOL_NONE	No security protocol has been selected. A secure connection has not been established with the server. The remaining member values in this structure are not valid and should be ignored.
SECURITY_PROTOCOL_SSH1	The Secure Shell 1.0 protocol has been selected. This protocol has been deprecated and is no longer widely used. It is not recommended that this protocol be used when establishing secure connections. This protocol can only be specified when connecting to an SSH server and is not supported with any other application protocol.
SECURITY_PROTOCOL_SSH2	The Secure Shell 2.0 protocol has been selected. This is the most commonly used version of the protocol. It is recommended that this version of the protocol be used unless the server explicitly requires the client to use an earlier version. This protocol can only be specified when connecting to an SSH server and is not supported with any other application protocol.

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
SECURITY_CIPHER_DES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher (also known as the Rijndael cipher) is a fixed block size cipher which use a key size of 128, 192 or 256 bits. This cipher is supported on Windows XP SP3 and later versions of the operating system.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected. One of the following values will be returned:

Constant	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected. One of the following values will be returned:

--	--

Constant	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography. This key exchange algorithm is only supported on Windows XP SP3 and later versions of the operating system.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server. This member only has meaning for connections using the SSL or TLS protocols.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertIssuer

A pointer to a string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertSubject

A pointer to a string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszFingerprint

A pointer to a string which contains a sequence of hexadecimal values that uniquely identify the server. This member is only used when a connection has been established using the Secure Shell (SSH) protocol.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

SSHOPTIONDATA Structure

This structure specifies additional option information for the client session. A pointer to this structure can be passed to the [Connect](#) method.

```
#define SSH_MAXTERMNAMELEN    32
#define SSH_MAXHOSTNAMELEN   128
#define SSH_MAXUSERNAMELEN   128
#define SSH_MAXPASSWORDLEN   128
#define SSH_MAXCOMMANDLEN    512

typedef struct _SSHOPTIONDATA
{
    DWORD dwSize;
    DWORD dwReserved;
    UINT nProxyType;
    UINT nProxyPort;
    TCHAR szProxyHost[SSH_MAXHOSTNAMELEN];
    TCHAR szProxyUser[SSH_MAXUSERNAMELEN];
    TCHAR szProxyPassword[SSH_MAXPASSWORDLEN];
    UINT nTermCols;
    UINT nTermRows;
    TCHAR szTermName[SSH_MAXTERMNAMELEN];
    TCHAR szCommandLine[SSH_MAXCOMMANDLEN];
} SSHOPTIONDATA, *LPSSHOPTIONDATA;
```

Members

dwSize

An unsigned integer value which specifies the size of the SSHOPTIONDATA structure. This member must be initialized prior to passing the structure to the Connect method.

dwReserved

An unsigned integer value that is reserved for internal use, and should always be initialized to a value of zero.

nProxyType

An unsigned integer value that specifies the type of proxy that the client should connect through. This structure member is only used if the option SSH_OPTION_PROXYSERVER has been specified. Possible values are:

Constant	Description
SSH_PROXY_NONE	No proxy server should be used when establishing the connection.
SSH_PROXY_HTTP	The connection should be established on port 80 using HTTP. An alternate port number can be specified by setting the <i>nProxyPort</i> structure member to the desired value.
SSH_PROXY_TELNET	The connection should be established on port 23 using TELNET. An alternate port number can be specified by setting the <i>nProxyPort</i> structure member to the desired value.

nProxyPort

An unsigned integer value that specifies the port number which should be used to establish the proxy connection. A value of zero specifies that the default port number appropriate for the

selected protocol should be used. This structure member is only used if the option SSH_OPTION_PROXYSERVER has been specified.

szProxyHost

A null terminated string which specifies the host name or IP address of the proxy server. This structure member is only used if the option SSH_OPTION_PROXYSERVER has been specified.

szProxyUser

A null terminated string which specifies the user name which is used to authenticate the connection through the proxy server. This structure member is only used if the option SSH_OPTION_PROXYSERVER has been specified.

szProxyPassword

A null terminated string which specifies the password which is used to authenticate the connection through the proxy server. This structure member is only used if the option SSH_OPTION_PROXYSERVER has been specified.

nTermCols

An unsigned integer value which specifies the number of columns for the virtual terminal allocated for the client session. The default number of columns is 80. This structure member is only used if the option SSH_OPTION_TERMINAL has been specified.

nTermRows

An unsigned integer value which specifies the number of rows for the virtual terminal allocated for the client session. The default number of rows is 25. This structure member is only used if the option SSH_OPTION_TERMINAL has been specified.

szTermName

A null terminated string which specifies the name of the terminal emulation type. On UNIX based systems, this name typically corresponds to an entry in the terminal capability database (either termcap or terminfo). If the name is not specified, then the default name terminal name of "unknown" will be used. This structure member is only used if the option SSH_OPTION_TERMINAL has been specified.

szCommandLine

A null terminated string which specifies the command that should be executed on the server. The output from the command is returned to the client, and the session is terminated. This structure member is only used if the option SSH_OPTION_COMMAND has been specified.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

Simple Mail Transfer Protocol Class Library

Submit email messages for delivery to one or more recipients.

Reference

- [Class Methods](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

Class Name	CSmtpClient
File Name	CSMTPV10.DLL
Version	10.0.1468.2518
LibID	D74CB488-F2C1-4F78-858A-201AEA7483FE
Import Library	CSMTPV10.LIB
Dependencies	None
Standards	RFC 821, RFC 1425, RFC 1869, RFC 2821

Overview

The Simple Mail Transfer Protocol (SMTP) enables applications to deliver email messages to one or more recipients. The class library provides an interface for addressing and delivering messages, and extended features such as user authentication and delivery status notification. Unlike Microsoft's Messaging API (MAPI) or Collaboration Data Objects (CDO), there is no requirement to have certain third-party email applications installed or specific types of servers installed on the local system. The CSmtpClient class can be used to deliver mail through a wide variety of systems, from standard UNIX based mail servers to Windows systems running Microsoft Exchange.

Using this class library, messages can be delivered directly to the recipient, or they can be routed through a relay server, such as an Internet service provider's mail system. The SocketTools CMailMessage class can be integrated with this class in order to provide an extremely simple, yet flexible interface for composing and delivering messages.

This class library supports secure connections using the standard SSL and TLS protocols. Both implicit and explicit SSL connections are supported, as well as client certificates used for authentication.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Simple Mail Transfer Protocol Class Methods

Class	Description
CSmtpClient	Constructor which initializes the current instance of the class
~CSmtpClient	Destructor which releases resources allocated by the class
Method	Description
AddRecipient	Add an address to the recipient list
AppendMessage	Append contents of specified file to the current message
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified client handle to another thread
Authenticate	Authenticate the client session with a user name and password
Cancel	Cancel the current blocking operation
CloseMessage	Close the message being composed and submit for delivery
Command	Send a command to the server
Connect	Establish a connection with a server
CreateMessage	Create a new message
CreateSecurityCredentials	Allocate a structure to establish client security credentials
DeleteSecurityCredentials	Delete the specified client security credentials
DetachHandle	Detach the handle for the current instance of this class
DisableEvents	Disable all event notification, including event callbacks
DisableTrace	Disable logging of socket function calls to the trace log
Disconnect	Disconnect from the current server
EnableEvents	Enable event handling by the library
EnableTrace	Enable logging of socket function calls to a file
ExpandAddress	Expand the specified address
FreezeEvents	Suspend and resume event handling by the client
GetCurrentDate	Return the current date and time
GetDeliveryOptions	Return the delivery options for the current session
GetErrorString	Return a description for the specified error code
GetExtendedOptions	Return the extended options supported by the server
GetHandle	Return the client handle used by this instance of the class
GetLastError	Return the last error code
GetLocalName	Return the local host name assigned for the client session
GetResultCode	Return the result code from the previous command

GetResultString	Return the result string from the previous command
GetSecurityInformation	Return security information about the current client connection
GetStatus	Return the current status of the client
GetTimeout	Return the number of seconds until an operation times out
GetTransferStatus	Return data transfer statistics
IsBlocking	Determine if the client is blocked, waiting for information
IsConnected	Determine if the client is connected to the server
IsInitialized	Determine if the class has been successfully initialized
IsReadable	Determine if data can be read from the server
IsWritable	Determine if data can be written to the server
RegisterEvent	Register an event handler for the specified event
Reset	Reset the client and return to a command state
SendMessage	Send message to the specified recipient
SetDeliveryOptions	Set the delivery options for the current session
SetLastError	Set the last error code
SetLocalName	Set the local host name to be used for the client session
SetTimeout	Set the number of seconds until an operation times out
ShowError	Display a message box with a description of the specified error
SmtpeventProc	Process events generated by the client
SubmitMessage	Compose and submit a message for delivery to the specified mail server
VerifyAddress	Verify that the specified address is valid
Write	Write data to the server

CSmtpClient::CSmtpClient Method

`CSmtpClient();`

The **CSmtpClient** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **SmtplibInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmtpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[~CSmtpClient](#), [IsInitialized](#)

CSmtpClient::~CSmtpClient

`~CSmtpClient();`

The **CSmtpClient** destructor releases resources allocated by the current instance of the **CSmtpClient** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CSmtpClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmtpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CSmtpClient](#)

CSmtpClient::AddRecipient Method

```
INT AddRecipient(  
    LPCTSTR lpszAddress  
);
```

The **AddRecipient** method adds the specified address to the recipient list for the current message. This method should be called once for each recipient.

Parameters

lpszAddress

Points to a string which specifies the address to be added to the recipient list.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CloseMessage](#), [CreateMessage](#), [ExpandAddress](#), [VerifyAddress](#)

CSmtpClient::AppendMessage Method

```
INT AppendMessage(  
    LPCTSTR lpszFileName  
);  
  
INT AppendMessage(  
    LPBYTE lpMessage,  
    DWORD dwMessageSize  
);  
  
INT AppendMessage(  
    HGLOBAL hgblMessage,  
    DWORD dwMessageSize  
);
```

The **AppendMessage** method writes the contents of a specified file or buffer to the data stream, appending it to the current message contents.

Parameters

lpszFileName

A pointer to a string which specifies the name of a file. The contents of the file are appended to the current message being submitted to the mail server for delivery.

lpMessage

Pointer to a buffer which contains the message data to be appended to the current message.

hgblMessage

A global memory handle which references data that is to be appended to the current message.

dwMessageSize

An unsigned integer which specifies the length of the message in bytes.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **AppendMessage** method is used to append the contents of a file, memory buffer, or global memory handle to the current message that is being composed for delivery. To send a complete RFC 822 formatted message, refer to the **SendMessage** method.

This method will cause the current thread to block until the complete message has been written, a timeout occurs or the operation is canceled. During the transfer, the SMTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of a transfer while it is in progress, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CloseMessage](#), [CreateMessage](#), [SendMessage](#)

CSmtpClient::AttachHandle Method

```
VOID AttachHandle(  
    HCLIENT hClient  
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle already has been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the *hClient* parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

See Also

[AttachThread](#), [DetachHandle](#), [GetHandle](#)

CSmtpClient::AttachThread Method

```
DWORD AttachThread(  
    DWORD dwThreadId  
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

See Also

[AttachHandle](#), [Cancel](#), [Connect](#), [DetachHandle](#), [Disconnect](#), [GetHandle](#)

CSmtpClient::Authenticate Method

```
INT Authenticate(  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    UINT nAuthType  
);
```

The **Authenticate** method provides client authentication information to the server.

Parameters

lpszUserName

A null terminated string which specifies the account name for the user authorized to send mail through the server.

lpszPassword

A null terminated string which specifies the password to be used when authenticating the current client session. If you are using the SMTP_AUTH_XOAUTH2 or SMTP_AUTH_BEARER authentication methods, this parameter is not a password, instead it specifies the bearer token provided by the mail service.

nAuthType

An integer value which specifies which method the library should use to authenticate the client session. This parameter should be set to one of the following values:

Constant	Description
SMTP_AUTH_LOGIN	The client will authenticate using the AUTH LOGIN command. This encodes the username and password, however the credentials are not encrypted and it is recommended you use a secure connection. This is the default method accepted by most mail servers and is the preferred authentication type for most clients.
SMTP_AUTH_PLAIN	The client will authenticate using the AUTH PLAIN command. This encodes the username and password, however the credentials are not encrypted and it is recommended you use a secure connection. The server must support the PLAIN Simple Authentication and Security Layer (SASL) mechanism as defined in RFC 4616.
SMTP_AUTH_XOAUTH2	The client will authenticate using the AUTH XOAUTH2 command. This authentication method does not require the user password, instead the <i>lpszPassword</i> parameter must specify the bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this method will fail.
SMTP_AUTH_BEARER	The client will authenticate using the AUTH OAUTHBEARER command as defined in RFC 7628. This authentication method does not require the user password, instead the <i>lpszPassword</i> parameter must specify the bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this method

will fail.

Return Value

If the method succeeds, the return value is the command result code. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

To submit a mail message for delivery, virtually all public mail servers require clients to authenticate and will only accept messages from authorized users. In some cases, they may also require the sender email address match the account being used to authenticate the session. It is also typical for most public mail servers to reject authentication attempts over a standard (non-secure) connection. You should always use a secure connection whenever possible.

All authentication methods require the mail server to support the standard service extensions for authentication as specified in the RFC 4954. The server must support the ESMTP protocol extensions and the AUTH command. A user name and password are required for authentication. If you wish to authenticate without a user password, you must use one of the OAuth 2.0 authentication methods.

If the *nAuthType* parameter is omitted, it will default to using SMTP_AUTH_LOGIN and this is accepted by most mail servers. It is common for mail servers to allow the SMTP_AUTH_PLAIN method as well, however it is recommended you explicitly check whether the server supports the desired authentication method by calling the **GetExtendedOptions** method. If you attempt to use an authentication type which is not supported by the server, this method will fail and the last error code will be set to ST_ERROR_INVALID_AUTHENTICATION_TYPE.

You should only use an OAuth 2.0 authentication method if you understand the process of how to request the access token. Obtaining an access token requires registering your application with the mail service provider (e.g.: Microsoft or Google), getting a unique client ID associated with your application and then requesting the access token using the appropriate scope for the service. Obtaining the initial token will typically involve interactive confirmation on the part of the user, requiring they grant permission to your application to access their mail account.

The SMTP_AUTH_XOAUTH2 and SMTP_AUTH_BEARER authentication methods are similar, but they are not interchangeable. Both use an OAuth 2.0 bearer token to authenticate the client session, but they differ in how the token is presented to the server. It is currently preferable to use the XOAUTH2 method because it is more widely available and some service providers do not yet support the OAUTHBEARER method.

Your application should not store an OAuth 2.0 bearer token for later use. They have a relatively short lifespan, typically about an hour, and are designed to be used with that session. You should specify offline access as part of the OAuth 2.0 scope if necessary and store the refresh token provided by the service. The refresh token has a much longer validity period and can be used to obtain a new bearer token when needed.

Example

```
DWORD dwOptions = 0;

// Determine which extended options and authentication methods
// are supported by this server

BOOL bExtended = pClient->GetExtendedOptions(&dwOptions);

if (bUseBearerToken)
{
```

```

    if (bExtended && (dwOptions & SMTP_EXTOPT_XOAUTH2))
    {
        INT nResult = pClient->Authenticate(lpszUserName, lpszBearerToken,
SMTP_AUTH_XOAUTH2);

        if (nResult == SMTP_ERROR)
        {
            // An error occurred during authentication; when using an
            // OAuth 2.0 bearer token, this typically means that the token
            // has expired and must be refreshed
            return;
        }
    }
    else
    {
        // The server does not support XOAUTH2
        return;
    }
}
else
{
    if (bExtended && (dwOptions & SMTP_EXTOPT_AUTHLOGIN))
    {
        INT nResult = pClient->Authenticate(lpszUserName, lpszPassword);

        if (nResult == SMTP_ERROR)
        {
            // An error occurred during authentication
            return;
        }
    }
    else
    {
        // The server does not support AUTH LOGIN
        return;
    }
}
}

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [GetExtendedOptions](#)

CSmtpClient::Cancel Method

```
INT Cancel();
```

The **Cancel** method cancels any outstanding blocking operation in the client, causing the blocking method to fail. The application may then retry the operation or terminate the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

See Also

[IsBlocking](#)

CSmtpClient::CloseMessage Method

```
INT CloseMessage();
```

The **CloseMessage** method ends the composition of the current message. The server then queues the message for delivery to each recipient specified by the client.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **CloseMessage** method should be called after all of the message data has been written to the data stream.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AppendMessage](#), [CreateMessage](#), [Write](#)

CSmtpClient::Command Method

```
INT Command(  
    LPCTSTR LpszCommand,  
    LPCTSTR LpszParameter  
);
```

The **Command** method sends a command to the server, and returns the result code back to the caller. This method is typically used for site-specific commands not directly supported by the API.

Parameters

LpszCommand

The command which will be executed by the server.

LpszParameter

An optional command parameter. If the command requires more than one parameter, then they should be combined into a single string, with a space separating each parameter. If the command does not accept any parameters, this value may be NULL.

Return Value

If the command was successful, the method returns the result code. If the command failed, the method returns SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

A list of valid commands can be found in the technical specification for the protocol. Many servers will list supported commands when the HELP command is used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetResultCode](#), [GetResultString](#)

CSmtpClient::Connect Method

```
BOOL Connect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **Connect** method is used to establish a connection with the server.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 25. An alternative port is 587, which is commonly used by authenticated clients to submit messages for delivery. For implicit SSL connections, the default port number is 465.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SMTP_OPTION_NONE	No additional options are specified when establishing a connection with the server. A standard, non-secure connection will be used and the client will not attempt to use extended (ESMTP) features of the protocol. Note that if the mail server requires authentication, the SMTP_OPTION_EXTENDED option must be specified.
SMTP_OPTION_EXTENDED	Extended SMTP commands should be used if possible. This option enables features such as authentication and delivery status notification. If this option is not specified, the library will not attempt to use any extended features. This option is automatically enabled if the connection is established on port 587 because submitting messages for delivery using this port typically requires client authentication.
SMTP_OPTION_TUNNEL	This option specifies that a tunneled TCP

	connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
SMTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
SMTP_OPTION_SECURE	This option specifies that a secure connection should be established with the server and requires that the server support either the SSL or TLS protocol. The client will initiate the secure session using the STARTTLS command.
SMTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using either the SSL or TLS protocol, and the secure session must be negotiated immediately after the connection has been established.
SMTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
SMTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
SMTP_OPTION_FREETHREAD	This option specifies that this instance of the class may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the class instance is synchronized across multiple threads.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous socket events that occur. If this argument is NULL, then the client session will be blocking and no network events will be sent to the client.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the *hEventWnd* argument is NULL, this argument should be specified as WM_NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The use of Windows event notification messages has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

To prevent this method from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **Connect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

If you specify an event notification window, then the client session will be asynchronous. When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
SMTP_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
SMTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
SMTP_EVENT_READ	Data is available to read by the client. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the calling process is in asynchronous mode.
SMTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
SMTP_EVENT_TIMEOUT	The client has timed out while waiting for a response from the server. Note that under some circumstances this event can be generated for a non-blocking connection, such as when the client is establishing a secure connection.
SMTP_EVENT_CANCEL	The client has canceled the current operation.
SMTP_EVENT_COMMAND	The client has processed a command that was sent to the server. The result code and result string can be used to determine if the

	response to the command. The high word of the IParam parameter should be checked, since this notification message will also be posed if the command cannot be executed.
SMTP_EVENT_PROGRESS	This event notification is sent periodically during lengthy blocking operations, such as retrieving a complete message from the server.

To cancel asynchronous notification and return the client to a blocking mode, use the **DisableEvents** method.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the class instance is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call methods using that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the SMTP_OPTION_FREETHREAD option enables any thread to call any method in that instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same instance.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Authenticate](#), [Disconnect](#), [GetExtendedOptions](#)

CSmtpClient::CreateMessage Method

```
INT CreateMessage(  
    LPCTSTR lpszSender,  
    DWORD dwMessageSize  
);
```

The **CreateMessage** method creates a new message for delivery.

Parameters

lpszSender

A pointer to a string which specifies the email address of the user sending the message. This typically corresponds to the address in the From header of the message, but it is not required that they be the same.

dwMessageSize

An unsigned integer which specifies the size of the message in bytes. If the size of the message is unknown, this value should be zero. This parameter is ignored if the server does not support extended features. If the message size is larger than what the server will accept, this method will fail. Most Internet Service Providers impose a limit on the size of an email message, typically between 5 and 10 megabytes.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **CreateMessage** method begins the composition of a new message to be submitted to the mail server for delivery. There are several steps that must be followed when dynamically composing a message for delivery:

1. Call the **CreateMessage** method to begin the message composition. The sender email address should generally be the same address as the one used in the "From" header field in the message.
2. Call the **AddRecipient** method for each recipient of the message. These addresses are typically specified in the "To" and "Cc" header fields in the message. Additional addresses may also be provided which are not specified in the email message itself. This is how one or more blind carbon copies of a message is delivered. Most servers have a limit on the total number of recipients that may be specified for a single message. This limit is usually around 100 addresses.
3. Call the **Write** method to write the contents of the message to the data stream. The application may also choose to use the **AppendMessage** method to write out a large amount of message data, or write the contents of a file to the data stream.
4. Call the **CloseMessage** method to close the message and submit it to the mail server for delivery.

For applications that do not need to dynamically compose the message and already have the message contents stored in a file or memory buffer, the **SendMessage** method is the preferred method of submitting a message for delivery.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AddRecipient](#), [AppendMessage](#), [CloseMessage](#), [SendMessage](#), [Write](#)

CSmtpClient::CreateSecurityCredentials Method

```
BOOL CreateSecurityCredentials(  
    DWORD dwProtocol,  
    DWORD dwOptions,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName  
);  
  
BOOL CreateSecurityCredentials(  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName  
);  
  
BOOL CreateSecurityCredentials(  
    LPCTSTR lpszCertName  
);
```

The **CreateSecurityCredentials** method establishes the security credentials for the client session.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols.

	This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpzUserName

A pointer to a string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A pointer to a string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A pointer to a string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Example

```
pClient->CreateSecurityCredentials(  
    SECURITY_PROTOCOL_DEFAULT,  
    0,  
    NULL,  
    NULL,  
    lpszCertStore,  
    lpszCertName);
```

```
bConnected = pClient->Connect(lpszHostName,  
                               SMTP_PORT_SECURE,  
                               SMTP_TIMEOUT,  
                               SMTP_OPTION_SECURE);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [DeleteSecurityCredentials](#), [GetSecurityInformation](#), [SECURITYCREDENTIALS](#)

CSmtpClient::DeleteSecurityCredentials Method

```
VOID DeleteSecurityCredentials();
```

The **DeleteSecurityCredentials** method releases the security credentials for the current session.

Parameters

None.

Return Value

None.

Remarks

This method can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated. The security credentials are released when the class destructor is called, so it is normally not required that the application explicitly call this method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateSecurityCredentials](#)

CSmtpClient::DetachHandle Method

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmtpv10.lib`

See Also

[AttachHandle](#), [GetHandle](#)

CSmtpClient::DisableEvents Method

```
INT DisableEvents();
```

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the client has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

See Also

[EnableEvents](#), [RegisterEvent](#)

CSmtpClient::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

See Also

[EnableTrace](#)

CSmtpClient::Disconnect Method

VOID Disconnect();

The **Disconnect** method terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

See Also

[Connect](#), [IsConnected](#)

CSmtpClient::EnableEvents Method

```
INT EnableEvents(  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Constant	Description
SMTP_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
SMTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
SMTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
SMTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking

	operation. This event is only generated if the client is in asynchronous mode.
SMTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
SMTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
SMTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
SMTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

It is not required that the client be placed in asynchronous mode in order to receive command and progress event notifications. To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

See Also

[DisableEvents](#), [RegisterEvent](#)

CSmtpClient::EnableTrace Method

```
BOOL EnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **EnableTrace** method enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace method logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableTrace](#)

CSmtpClient::ExpandAddress Method

```
INT ExpandAddress(  
    LPCTSTR lpszMailingList,  
    LPTSTR lpszAddresses,  
    INT nMaxLength  
);
```

```
INT ExpandAddress(  
    LPCTSTR lpszMailingList,  
    CString& strAddresses  
);
```

The **ExpandAddress** method expands the specified mailing list, returning the membership of that list.

Parameters

lpszMailingList

Points to a string which specifies the mailing list that the server should expand into full addresses.

lpszAddresses

Points to a buffer that the expanded addresses will be copied into. This argument may also be a **CString** object which will contain the expanded addresses when the method returns.

nMaxLength

Maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **ExpandAddress** method requests that the server expand the specified email address. Typically this is used to expand aliases which refer to a mailing list, returning all of the members of that list. A server may not support this command, or may restrict its usage. An application should not depend on the ability to expand addresses.

This method cannot be called while a mail message is being composed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AddRecipient](#), [VerifyAddress](#)

CSmtpClient::FreezeEvents Method

```
INT FreezeEvents(  
    BOOL bFreeze  
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the client.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableEvents](#), [EnableEvents](#), [RegisterEvent](#)

CSmtpClient::GetCurrentDate Method

```
INT GetCurrentDate(  
    LPTSTR lpszDateString,  
    INT nMaxLength  
);  
  
INT GetCurrentDate(  
    CString& strDateString  
);
```

The **GetCurrentDate** method copies the current date and time to the specified buffer in a format that is commonly used in mail messages. This date format should be used in all date-related fields in the message header.

Parameters

lpszDateString

Pointer to a string buffer that will contain the current date and time when the method returns. This argument may also be a **CString** object.

nMaxLength

The maximum number of characters that can be copied into the string buffer.

Return Values

If the method succeeds, the return value is the number of characters copied into the buffer, not including the null-terminator. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The date value that is returned is adjusted for the local timezone.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CloseMessage](#), [CreateMessage](#)

CSmtpClient::GetDeliveryOptions Method

```
BOOL GetDeliveryOptions(  
    LPDWORD LpdwOptions  
);
```

The **GetDeliveryOptions** method returns the delivery status notification options for the current session.

Parameters

LpdwOptions

Address of a variable that will be set to the current delivery options. This bitmask is created by combining one or more of the following values with a bitwise Or operator:

Constant	Description
SMTP_NOTIFY_NEVER	Never return information about the success or failure of the message delivery process.
SMTP_NOTIFY_SUCCESS	Return a message to the sender if the message has been successfully delivered to the recipient's mail server.
SMTP_NOTIFY_FAILURE	Return a message to the sender if the message could not be delivered to the recipient's mail server.
SMTP_NOTIFY_DELAY	Return a message to the sender if delivery of the message was delayed.
SMTP_RETURN_HEADERS	Return only the message headers to the sender.
SMTP_RETURN_MESSAGE	Return the complete message headers and body to the sender.

Return Values

If the method succeeds, the return value is a non-zero value. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetDeliveryOptions** method returns the current delivery options for the client session. Note that delivery options are only available on those mail servers which support delivery status notification (DSN) using the extended SMTP protocol. The client must connect specifying SMTP_OPTION_EXTENDED in order to use extended server options.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

See Also

[Connect](#), [GetExtendedOptions](#), [SetDeliveryOptions](#)

CSmtpClient::GetErrorString Method

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);
```

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    CString& strDescription  
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLastError](#), [SetLastError](#), [ShowError](#)

CSmtpClient::GetExtendedOptions Method

```
BOOL GetExtendedOptions(  
    LPDWORD LpdwOptions  
);
```

The **GetExtendedOptions** method returns the extended server options for the current session.

Parameters

LpdwOptions

Address of a variable that will be set to the current server options. This bitmask is created by combining one or more of the following values with a bitwise Or operator:

Constant	Description
SMTP_EXTOPT_EXPN	The server supports address expansion using the EXPN command. The ExpandAddress method can be used to expand addresses, which typically returns the email addresses associated with a mailing list. Most public mail servers restrict or disable this functionality because it can present a security risk. If a server does permit the use of the command, it is often limited to specific authorized users.
SMTP_EXTOPT_VRFY	The server supports verification of addresses using the VRFY command. The VerifyAddress method can be used to verify addresses. Most public mail servers restrict the ability for clients to verify email addresses to prevent potential abuse. If a server does permit the use of the command, it is often limited to specific authorized users.
SMTP_EXTOPT_DSN	The server supports delivery status notification (DSN) which allows the sender to be notified when a message has been delivered, or when an error occurs during the delivery process. The SetDeliveryOptions method can be used to specify the delivery options to be used in the current session.
SMTP_EXTOPT_SIZE	The server supports the use of the SIZE parameter, which enables the client to determine the maximum message size that may be delivered through the server. Most public mail servers impose a limit of on the total size of a message, including any encoded attachments.
SMTP_EXTOPT_ETRN	The server supports the use of the ETRN command, instructing the server to start processing its message queues for a specific host. Most public mail servers do not support this capability and its use has been deprecated.
SMTP_EXTOPT_8BITMIME	The server supports the delivery of messages that contain characters with the high bit set. Most servers support this option, however it is recommended that you encode any message text which contains non-ASCII characters to ensure the broadest compatibility with other servers and clients.

SMTP_EXTOPT_STARTTLS	The server supports explicit TLS sessions. This extended option is used internally to determine how secure connections should be established, and if a secure connection can be made using the standard submission port.
SMTP_EXTOPT_UTF8	The server supports UTF-8 encoding in email addresses and the message envelope. Not all mail servers will have this extended capability enabled, and applications should not depend on being able to provide internationalized user and domain names unless this option bitflag has been set.

In addition, there are extended options which specify the authentication methods supported by the server. A server will typically support multiple authentication methods and may be one or more of the following values:

Constant	Description
SMTP_EXTOPT_AUTHLOGIN	The server supports client authentication using the AUTH LOGIN command. This is the default authentication method and is supported by most mail servers. The user name and password are encoded in a specific format, but are not encrypted. The client should use a secure connection whenever possible.
SMTP_EXTOPT_AUTHPLAIN	The server supports client authentication using the AUTH PLAIN command. The use name and password are encoded in a specific format, but are not encrypted. If a server supports this authentication method, it is very likely it also supports AUTH LOGIN. It is recommended you use only use AUTH PLAIN authentication if the server does not support AUTH LOGIN.
SMTP_EXTOPT_XOAUTH2	The server supports client authentication using AUTH XOAUTH2 command. Instead of a password, an OAuth 2.0 bearer token is used to authenticate the user which previously authorized access to the mail server using their account information. The connection must be secure to use this authentication method.
SMTP_EXTOPT_BEARER	The server supports client authentication using AUTH OAUTHBEARER command as specified in RFC 7628. Instead of a password, an OAuth 2.0 bearer token is used to authenticate the user which previously authorized access to the mail server using their account information. The connection must be secure to use this authentication method. The connection must be secure to use this authentication method.

Return Value

If the method succeeds, the return value is a non-zero value. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetExtendedOptions** method returns the extended options supported by the server. The use of extended options requires that the server support the ESMTP protocol, and that the client connect using the SMTP_OPTION_EXTENDED option.

You should check these options prior to calling **Authenticate** to determine which authentication methods are acceptable to the server. If you wish to use an OAuth 2.0 bearer token, always check to make sure either the SMTP_EXTOPT_XOAUTH2 or SMTP_EXTOPT_BEARER bitflags are set in the options value returned by this method.

Example

```
BOOL bExtended = FALSE;
DWORD dwOptions = 0;

// Determine which extended options and authentication methods
// are supported by this server

bExtended = pClient->GetExtendedOptions(&dwOptions);

if (bExtended && (dwOptions & SMTP_EXTOPT_XOAUTH2))
{
    INT nResult = pClient->Authenticate(lpszUserName, lpszBearerToken,
SMTP_AUTH_XOAUTH2);

    if (nResult == SMTP_ERROR)
    {
        // An error occurred during authentication; when using an
        // OAuth 2.0 bearer token, this typically means that the token
        // has expired and must be refreshed
        return;
    }
}
else
{
    // The server does not support XOAUTH2
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

See Also

[Authenticate](#), [Connect](#), [GetDeliveryOptions](#), [SetDeliveryOptions](#)

CSmtpClient::GetHandle Method

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmtpv10.lib`

See Also

[AttachHandle](#), [DetachHandle](#), [IsInitialized](#)

CSmtpClient::GetLastError Method

```
DWORD GetLastError();  
  
DWORD GetLastError(  
    CString& strDescription  
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError(0)** when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or SMTP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

See Also

[GetErrorString](#), [SetLastError](#), [ShowError](#)

CSmtpClient::GetLocalName Method

```
INT GetLocalName(  
    LPTSTR lpszLocalName,  
    INT nMaxLength  
);  
  
INT GetLocalName(  
    CString& strLocalName  
);
```

The **GetLocalName** method returns the local domain name used to identify the client to the mail server.

Parameters

lpszLocalName

A pointer to the buffer that will contain the local domain name as a string. This may also be a **CString** object which will contain the domain name when the method returns.

cbDescription

The maximum number of characters that may be copied into the string buffer.

Return Value

If the method succeeds, the return value is the length of the domain name string. If the method fails, the return value is zero, meaning that no local domain name has been specified for this client session.

Remarks

If no local domain name has been explicitly set, then the client will use the default domain name for the local host as configured in the operating system. Note that this is not a general purpose method which can be used to determine the domain name that has been assigned to the local host. It will only return the local domain name set by a previous call to the **SetLocalName** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [SetLocalName](#)

CSmtpClient::GetResultCode Method

INT GetResultCode();

The **GetResultCode** method reads the result code returned by the server in response to a command. The result code is a three-digit numeric code, and indicates if the operation succeeded, failed or requires additional action by the client.

Parameters

None.

Return Value

If the method succeeds, the return value is the result code. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

Result codes are three-digit numeric values returned by the server. They may be broken down into the following ranges:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

See Also

[Command](#), [GetResultString](#)

CSmtpClient::GetResultString Method

```
INT GetResultString(  
    LPTSTR lpszResult,  
    INT cbResult  
);
```

```
INT GetResultString(  
    CString& strResult  
);
```

The **GetResultString** method returns the last message sent by the server along with the result code.

Parameters

lpszResult

A pointer to the buffer that will contain the result string returned by the server. An alternate form of the method accepts a **CString** argument which will contain the result string returned by the server.

cbResult

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetResultString** method is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **GetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Command](#), [GetResultCode](#)

CSmtpClient::GetSecurityInformation Method

```
BOOL GetSecurityInformation(  
    LPSECURITYINFO LpSecurityInfo  
);
```

The **GetSecurityInformation** method returns security protocol, encryption and certificate information about the current client connection.

Parameters

LpSecurityInfo

A pointer to a [SECURITYINFO](#) structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;  
securityInfo.dwSize = sizeof(SECURITYINFO);  
  
if (pClient->GetSecurityInformation(&securityInfo))  
{  
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)  
    {  
        MessageBox(NULL, _T("The connection is not secure"),  
                    _T("Connection"), MB_OK);  
    }  
    else  
    {  
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)  
        {  
            MessageBox(NULL, _T("The connection is secure"),  
                        _T("Connection"), MB_OK);  
        }  
        else  
        {  
            MessageBox(NULL, _T("The server certificate not valid"),  
                        _T("Connection"), MB_OK);  
        }  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [CreateSecurityCredentials](#), [SECURITYINFO](#)

CSmtpClient::GetStatus Method

INT GetStatus();

The **GetStatus** method the current status of the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the client status code. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetStatus** method returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Constant	Description
0	SMTP_STATUS_UNUSED	No connection has been established.
1	SMTP_STATUS_IDLE	The client is current idle and not sending or receiving data.
2	SMTP_STATUS_CONNECT	The client is establishing a connection with the server.
3	SMTP_STATUS_READ	The client is reading data from the server.
4	SMTP_STATUS_WRITE	The client is writing data to the server.
5	SMTP_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

See Also

[IsBlocking](#), [IsConnected](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#)

CSmtpClient::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

See Also

[Connect](#), [IsReadable](#), [IsWritable](#), [Read](#), [SetTimeout](#), [Write](#)

CSmtpClient::GetTransferStatus Method

```
INT GetTransferStatus(  
    HCLIENT hClient,  
    LPSMTPTRANSFERSTATUS lpStatus  
);
```

The **GetTransferStatus** method returns information about the message being submitted to the mail server.

Parameters

hClient

Handle to the client session.

lpStatus

A pointer to an [SMTPTRANSFERSTATUS](#) structure which contains information about the status of the message being submitted for delivery.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetTransferStatus** method returns information about the current message being submitted, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no message currently being submitted, this method will return the status of the last successful submission made by the client.

In a multithreaded application, any thread in the current process may call this method to obtain the status of a submission for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableEvents](#), [GetStatus](#), [RegisterEvent](#)

CSmtpClient::IsBlocking Method

BOOL IsBlocking();

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking method may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Cancel](#), [GetStatus](#), [IsConnected](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#), [Read](#), [Write](#)

CSmtpClient::IsConnected Method

```
BOOL IsConnected();
```

The **IsConnected** method is used to determine if the client is currently connected to a server.

Parameters

None.

Return Value

If the client is connected to a server, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#)

CSmtpClient::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

See Also

[CSmtpClient](#), [IsBlocking](#), [IsConnected](#)

CSmtpClient::IsReadable Method

```
BOOL IsReadable(  
    INT nTimeout,  
    LPDWORD lpdwAvail  
);
```

The **IsReadable** method is used to determine if data is available to be read from the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the method returns a non-zero value. If the client cannot read any data, the method returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsConnected](#), [IsInitialized](#), [IsWritable](#), [Write](#)

CSmtpClient::IsWritable Method

```
BOOL IsWritable(  
    INT nTimeout  
);
```

The **IsWritable** method is used to determine if data can be written to the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the method returns a non-zero value. If the client cannot write any data, the method returns zero.

Remarks

Although this method can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsConnected](#), [IsInitialized](#), [IsReadable](#), [Write](#)

CSmtpClient::Read Method

```
INT Read(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Read(  
    CString& strBuffer,  
    INT cbBuffer  
);
```

The **Read** method reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be copied. An alternate form of this method allows a **CString** variable to be passed and data read from the socket will be returned in that string.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

When **Read** is called and the client is in non-blocking mode, it is possible that the method will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmtpv10.lib

See Also

[EnableEvents](#), [IsBlocking](#), [IsReadable](#), [IsWritable](#), [RegisterEvent](#), [Write](#)

CSmtpClient::RegisterEvent Method

```
INT RegisterEvent(  
    UINT nEventId,  
    SMTPEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Constant	Description
SMTP_EVENT_CONNECT	The connection to the server has completed.
SMTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
SMTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
SMTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
SMTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
SMTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
SMTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
SMTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **SmtplibEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **SmtplibEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This method is typically used to register an event handler that is invoked while a message is being submitted to the server for delivery. The SMTP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one SMTP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmtplib10.lib

See Also

[DisableEvents](#), [EnableEvents](#), [FreezeEvents](#), [SmtplibEventProc](#)

CSmtpClient::Reset Method

```
INT Reset();
```

The **Reset** method resets the client state and resynchronizes with the server. This method is typically called after an unexpected error has occurred, or an operation has been canceled.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The client cannot be reset while in a blocked state.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Cancel](#), [CloseMessage](#), [CreateMessage](#), [IsBlocking](#)

CSmtpClient::SendMessage Method

```
INT SendMessage(  
    LPCTSTR lpszFrom,  
    LPCTSTR lpszRecipient,  
    LPCTSTR lpszFileName  
);
```

```
INT SendMessage(  
    LPCTSTR lpszFrom,  
    LPCTSTR lpszRecipient,  
    LPBYTE lpMessage,  
    DWORD dwMessageSize  
);
```

```
INT SendMessage(  
    LPCTSTR lpszFrom,  
    LPCTSTR lpszRecipient,  
    HGLOBAL hglbMessage,  
    DWORD dwMessageSize  
);
```

The **SendMessage** method sends the contents of a file or buffer to the specified recipients.

Parameters

lpszFrom

Pointer to a string which specifies the email address of the sender.

lpszRecipient

Pointer to a string which specifies the recipient of the message. Multiple recipients may be specified by separating each address with a comma.

lpszFileName

Pointer to a string which specifies a file that contains the message to be delivered.

lpMessage

Pointer to a buffer which contains the message to be delivered. This may also be a global memory handle which references the message data.

dwMessageSize

An unsigned integer which specifies the length of the message in bytes.

Return Value

If the method succeeds, the return value is the result code from the server. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **SendMessage** method is used to send the contents of a file, buffer or global memory handle to the specified recipients. The message must be in the standard format as described in RFC 822. The **CMailMessage** class can be used to compose and export a message in the correct format.

This protocol is only concerned with the delivery of a message and not its contents. Header fields in the message are not parsed to determine the recipients. This recipient parameter should be a concatenation of all recipients, including carbon copies and blind carbon copies, with each address separated with a comma.

This method will cause the current thread to block until the complete message has been delivered, a timeout occurs or the operation is canceled. During the transfer, the SMTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **EnableEvents**, or by registering a callback function using the **RegisterEvent** method.

To determine the current status of the transaction while it is in progress, use the **GetTransferStatus** method.

An alternative approach to creating a message without using the **CMailMessage** class is the **SubmitMessage** method. It accepts two structure parameters which define the message contents and the connection information for the mail server. This enables the application to compose the message and submit it for delivery in a single method call.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AppendMessage](#), [CloseMessage](#), [GetTransferStatus](#), [SubmitMessage](#)

CSmtpClient::SetDeliveryOptions Method

```
BOOL SetDeliveryOptions(  
    DWORD dwOptions  
);
```

The **SetDeliveryOptions** method sets the delivery status notification options for the current session.

Parameters

dwOptions

A bitmask that defines the current delivery options. This value is created by combining one or more of the following constants with a bitwise Or operator:

Constant	Description
SMTP_NOTIFY_NEVER	Never return information about the success or failure of the message delivery process.
SMTP_NOTIFY_SUCCESS	Return a message to the sender if the message has been successfully delivered to the recipient's mail server.
SMTP_NOTIFY_FAILURE	Return a message to the sender if the message could not be delivered to the recipient's mail server.
SMTP_NOTIFY_DELAY	Return a message to the sender if delivery of the message was delayed.
SMTP_RETURN_HEADERS	Return only the message headers to the sender.
SMTP_RETURN_MESSAGE	Return the complete message headers and body to the sender.

Return Value

If the method succeeds, the return value is a non-zero value. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetDeliveryOptions** method sets the current delivery options for the client session. Note that delivery options are only available on those mail servers which support delivery status notification (DSN) using the extended SMTP protocol. The client must connect specifying SMTP_OPTION_EXTENDED in order to use extended server options.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetDeliveryOptions](#), [GetExtendedOptions](#)

CSmtpClient::SetLastError Method

```
VOID SetLastError(  
    DWORD dwErrorCode  
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or SMTP_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

See Also

[GetErrorString](#), [GetLastError](#), [ShowError](#)

CSmtpClient::SetLocalName Method

```
BOOL SetLocalName(  
    LPCTSTR lpszLocalName  
);
```

The **SetLocalName** method sets the local domain name used to identify the client to the mail server.

Parameters

lpszLocalName

A pointer to a string which specifies the local domain name to be used by the client when establishing a connection with the mail server. A value of NULL or an empty string clears the domain name.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetLocalName** method should only be used to specify a domain name if it is absolutely necessary. In most cases, it is preferable to allow the library to automatically determine the correct domain name to use. Providing an invalid domain name may cause the mail server to reject the connection.

This method may only be called prior to establishing a connection with the mail server using the **Connect** method. After the connection has been made, this method will fail. To change the local domain name used by the client, you must first terminate the current session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [GetLocalName](#)

CSmtpClient::SetTimeout Method

```
INT SetTimeout(  
    UINT nTimeout  
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmtpv10.lib

See Also

[Connect](#), [GetTimeout](#), [IsReadable](#), [IsWritable](#), [Read](#), [Write](#)

CSmtpClient::ShowError Method

```
INT ShowError(  
    LPCTSTR lpszAppTitle,  
    UINT uType,  
    DWORD dwErrorCode  
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A pointer to a string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetErrorString](#), [GetLastError](#), [SetLastError](#)

CSmtpClient::SubmitMessage Method

```
BOOL SubmitMessage(  
    LPSMTPSERVER LpServer,  
    LPSMTPMESSAGE LpMessage,  
    SMTPEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

```
BOOL SubmitMessage(  
    LPSMTPSERVER LpServer,  
    LPSMTPMESSAGEEX LpMessage,  
    SMTPEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

The **SubmitMessage** method composes and submits a message for delivery to the specified mail server.

Parameters

lpServer

A pointer to an **SMTPSERVER** structure that contains information about the mail server that the message will be submitted to for delivery. This parameter cannot be NULL and the structure members must be properly initialized prior to calling this function.

lpMessage

A pointer to an **SMTPMESSAGE** or **SMTPMESSAGEEX** structure that contains information about the message, including the sender, recipients and the body of the message. This parameter cannot be NULL and the structure members must be properly initialized prior to calling this function.

lpEventProc

An optional pointer to the procedure-instance address of an application defined callback function. For more information about event handling and the callback function, see the description of the **SmtplibEventProc** callback function. If this parameter is NULL or is omitted, event notification is disabled.

dwParam

An optional user-defined integer value that is passed to the callback function. If the *lpEventProc* parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **SmtplibGetLastError**.

Remarks

The **SubmitMessage** method provides a high-level interface that enables an application to send an email message with a single function call. The **SMTPSERVER** and **SMTPMESSAGE** structures are used to provide the function with information about the mail server that will accept the message and the contents of the message itself. Note that this method does not require you to call the **Connect** method prior to calling this function.

If you need to specify additional custom headers in the message that is submitted for delivery, you should use the **SMTPMESSAGEEX** structure. It is an extended version of the message structure

which will allow you to define custom headers to be included in the message.

This method will cause the calling thread to block until the message has been submitted for delivery, an error occurs or the connection to the mail server times out. If an event handler is specified, then the callback function will be periodically invoked as the message is being sent. For large messages, the SMTP_EVENT_PROGRESS event can be used to monitor the submission process and update the user interface. The **GetTransferStatus** method can be used within the callback function to obtain information about the current status of the submission.

Example

```
CSmtpClient smtpClient;

SMTPSERVER mailServer;
ZeroMemory(&mailServer, sizeof(mailServer));
mailServer.lpszHostName = _T("smtp.gmail.com");
mailServer.nHostPort = SMTP_PORT_SUBMIT;
mailServer.lpszUserName = m_strSender;
mailServer.lpszPassword = m_strPassword;
mailServer.dwOptions = SMTP_OPTION_SECURE;

SMTPMESSAGE mailMessage;
ZeroMemory(&mailMessage, sizeof(mailMessage));
mailMessage.lpszFrom = m_strSender;
mailMessage.lpszTo = m_strRecipients;
mailMessage.lpszSubject = m_strSubject;
mailMessage.lpszText = m_strMessage;

if (smtpClient.SubmitMessage(&mailServer, &mailMessage))
    _tprintf(_T("SubmitMessage was successful\n"));
else
{
    CString strError;
    smtpClient.GetErrorString(strError);
    _tprintf(_T("SubmitMessage failed: %s\n"), strError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SendMessage](#), [SmtpEventProc](#), [SMTPMESSAGE](#), [SMTPMESSAGEEX](#), [SMTPSERVER](#)

CSmtpClient::VerifyAddress Method

```
INT VerifyAddress(  
    LPCTSTR lpszAddress,  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);
```

```
INT VerifyAddress(  
    LPCTSTR lpszAddress,  
    CString& strBuffer  
);
```

The **VerifyAddress** method verifies the specified address is valid.

Parameters

lpszAddress

Points to a string which specifies the address that the server should verify.

lpszBuffer

Points to a buffer that the verified address will be copied into. This argument may also be a **CString** object which will contain the verified address when the method returns.

nMaxLength

Maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the server result code. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **VerifyAddress** method requests that the server verify the specified email address. Typically this is used to verify that a recipient address is valid, and return a fully qualified email address for that recipient. A server may not support this command, or may restrict its usage. An application should not depend on the ability to verify addresses.

This method cannot be called while a mail message is being composed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AddRecipient](#), [ExpandAddress](#)

CSmtpClient::Write Method

```
INT Write(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Write(  
    LPCTSTR lpszBuffer  
    INT cbBuffer  
);
```

The **Write** method sends the specified number of bytes to the server.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server. In an alternate form of the method, the pointer is to a string.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero, unless a pointer to a string buffer is passed as the parameter. In that case, if the value is -1, all of the characters in the string, up to but not including the terminating null character, will be sent to the server.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is SMTP_ERROR. To get extended error information, call **GetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the next asynchronous notification message before it resumes sending data.

If the **Write** method is used to send the message contents to the server, the application must first call the **CreateMessage** method to specify the sender and the length of the message, followed by one or more calls to the **AddRecipient** method to specify each recipient of the message. When all of the message text has been submitted to the server, the application must call the **CloseMessage** method.

The message text is filtered by the **Write** method, and it will automatically normalize end-of-line character sequences to ensure the message meets the protocol requirements. The message itself must be in a standard RFC 822 or multi-part MIME message format, or the server may reject the message. Binary data, such as file attachments, should always be encoded. The **CMailMessage** class can be used to compose and export a message in the correct format, which can then be submitted to the server.

It is recommended that most applications use the **SendMessage** method to submit the message for delivery.

An alternative approach to creating a message without using the **CMailMessage** class is the **SubmitMessage** method. It accepts two structure parameters which define the message contents and the connection information for the mail server. This enables the application to compose the

message and submit it for delivery in a single method call.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AddRecipient](#), [AppendMessage](#), [CloseMessage](#), [CreateMessage](#), [SendMessage](#), [SubmitMessage](#)

Simple Message Transfer Protocol Data Structures

- SECURITYCREDENTIALS
- SECURITYINFO
- SMTPMESSAGE
- SMTPMESSAGEEX
- SMTPSERVER
- SMTPTRANSFERSTATUS
- SYSTEMTIME

SECURITYCREDENTIALS Structure

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on

	what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that

will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A pointer to a null terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A pointer to a null terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A pointer to a null terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A pointer to a null terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
------------	-------------

CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
----	---

MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
----	--

ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.
------	--

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

A pointer to a null terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used for SSH connections

and should always be NULL when establishing a secure connection using SSL or TLS.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* member should be set to CREDENTIAL_STORE_FILENAME and the *lpzCertStore* member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS#12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the *lpzPassword* member or the library will be unable to access the certificate.

Note that the *lpzUserName* and *lpzPassword* members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

The TLS 1.1 and TLS 1.2 protocols are only supported on Windows 7, Windows Server 2008 R2 and later versions of the platform. If these options are specified and the application is running on Windows XP or Windows Vista, the protocol version will be downgraded to TLS 1.0 for backwards compatibility with those platforms.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

SECURITYINFO Structure

This structure contains information about a secure connection that has been established.

```
typedef struct _SECURITYINFO
{
    DWORD        dwSize;
    DWORD        dwProtocol;
    DWORD        dwCipher;
    DWORD        dwCipherStrength;
    DWORD        dwHash;
    DWORD        dwHashStrength;
    DWORD        dwKeyExchange;
    DWORD        dwCertStatus;
    SYSTEMTIME   stCertIssued;
    SYSTEMTIME   stCertExpires;
    LPCTSTR      lpszCertIssuer;
    LPCTSTR      lpszCertSubject;
    LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to **sizeof(SECURITYINFO)** prior to passing the address of this structure to the function. Note that if this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the protocol that was selected to establish the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The

	<p>correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.</p>
SECURITY_PROTOCOL_TLS10	<p>The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS11	<p>The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS12	<p>The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.</p>

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_CIPHER_RC2	The RC2 block cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
SECURITY_CIPHER_DES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher, using 56-bit

	keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher (also known as the Rijndael cipher) is a fixed block size cipher which use a key size of 128, 192 or 256 bits. This cipher is supported on Windows XP SP3 and later versions of the operating system.
SECURITY_CIPHER_SKIPJACK	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
SECURITY_CIPHER_BLOWFISH	The Blowfish block cipher was selected. This is a variable key length cipher up to 448 bits, using a 64-bit block size.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected. One of the following values will be returned:

Constant	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected. One of the following values will be returned:

--	--

Constant	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography. This key exchange algorithm is only supported on Windows XP SP3 and later versions of the operating system.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server. This member only has meaning for connections using the SSL or TLS protocols. One of the following values will be returned:

Constant	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertIssuer

A pointer to a string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertSubject

A pointer to a string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszFingerprint

A pointer to a string which contains a sequence of hexadecimal values that uniquely identify the server. This member is only used when a connection has been established using the Secure Shell (SSH) protocol.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Unicode: Implemented as Unicode and ANSI versions.

SMTPMESSAGE Structure

This structure provides information about the contents of a message and is used by the [SubmitMessage](#) method.

```
typedef struct _SMTPMESSAGE
{
    LPCTSTR lpszFrom;
    LPCTSTR lpszTo;
    LPCTSTR lpszCc;
    LPCTSTR lpszBcc;
    LPCTSTR lpszSubject;
    LPCTSTR lpszText;
    LPCTSTR lpszHTML;
    LPCTSTR lpszAttach;
    UINT nCharSet;
    UINT nEncType;
    DWORD dwReserved;
} SMTPMESSAGE, *LPSMTPMESSAGE;
```

Members

lpszFrom

A pointer to a string that specifies the email address of the person sending the message. This structure member must point to a valid address and cannot be NULL.

lpszTo

A pointer to a string that specifies the email addresses of one or more recipients. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member must point to at least one valid address and cannot be NULL.

lpszCc

A pointer to a string that specifies the email addresses of one or more recipients that will receive copies of the message. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member may be NULL or point to an empty string.

lpszBcc

A pointer to a string that specifies the email addresses of one or more recipients that will receive blind copies of the message. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member may be NULL or point to an empty string. Unlike the recipients specified by the *lpszTo* and *lpszCc* members, any addresses specified by this member will not be included in the header of the email message.

lpszSubject

A pointer to a string that specifies the subject of the message. This structure member may be NULL, in which case no subject will be included in the message.

lpszText

A pointer to a string which contains the body of the message as plain text. Each line of text contained in the string should be terminated with a carriage-return and linefeed (CRLF) pair, which is recognized as the end-of-line. If this structure member is NULL or points to an empty string, then the *lpszHTML* member must specify the body of the message.

lpszHTML

A pointer to a string which contains the message using HTML formatting. If the *lpszText* member is not NULL, then a multipart message will be created with both plain text and HTML

text as the alternative. This allows mail clients to select which message body they wish to display. If the *lpszText* member is NULL or points to an empty string, then the message will only contain HTML. Although this is supported, it is not recommended because older mail clients may be unable to display the message correctly.

lpszAttach

A pointer to a string which specifies one or more file attachments for the message. If multiple files are to be attached to the message, each file name must be separated by a semi-colon. It is recommended that you provide the complete path to the file. If this structure member is NULL or points to an empty string, the message will be created without attachments.

nCharSet

A integer value which specifies the character set to use when composing the message. A value of zero specifies that the default USASCII character set should be used. The following values may also be used:

Constant	Description
MIME_CHARSET_USASCII	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the first bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_CHARSET_ISO8859_1	An 8-bit character set for most western European languages such as English, French, Spanish and German. This character set is also commonly referred to as Latin1.
MIME_CHARSET_ISO8859_2	An 8-bit character set for most central and eastern European languages such as Czech, Hungarian, Polish and Romanian. This character set is also commonly referred to as Latin2.
MIME_CHARSET_ISO8859_5	An 8-bit character set for Cyrillic languages such as Russian, Bulgarian and Serbian.
MIME_CHARSET_ISO8859_6	An 8-bit character set for Arabic languages. Note that the application is responsible for displaying text that uses this character set. In particular, any display engine needs to be able to handle the reverse writing direction and analyze the context of the message to correctly combine the glyphs.
MIME_CHARSET_ISO8859_7	An 8-bit character set for the Greek language.
MIME_CHARSET_ISO8859_8	An 8-bit character set for the Hebrew language. Note that similar to Arabic, Hebrew uses a reverse writing direction. An application which displays this character should be capable of processing bi-directional text where a single message may include both right-to-left and left-to-right languages, such as Hebrew and English.
MIME_CHARSET_ISO8859_9	An 8-bit character set for the Turkish language. This character set is also commonly referred to as Latin5.

nEncType

A numeric identifier which specifies the encoding type to use when composing the message. A value of zero specifies that default 7bit encoding should be used. The following values may also be used:

Constant	Description
MIME_ENCODING_7BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the most significant bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_ENCODING_8BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long and all bits are used. 8-bit encoding is typically used with multibyte character sets and is the default encoding used with Unicode text.
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.

dwReserved

An unsigned integer value reserved for future use. This structure member should always have a value of zero.

Remarks

This structure is used to define the contents of a message that will be submitted for delivery using the **SmtplibSubmitMessage** function. It is required that you specify a sender, at least one recipient and a message body. All other structure members may be NULL or have a value of zero to indicate that either the value is not required, or that a default should be used. It is recommended that you initialize all of the structure members to a value of zero using the **ZeroMemory** function prior to populating the structure.

email addresses may be specified as simple addresses, or as commented addresses that include the sender's name or other information. For example, any one of these address formats are acceptable:

```
user@domain.tld
User Name <user@domain.tld>
user@domain.tld (User Name)
```

To specify multiple addresses, you should separate each address by a comma or semi-colon. Note that the *lpszFrom* member cannot specify multiple addresses, however it is permitted with the *lpszTo*, *lpszCc* and *lpszBcc* structure members. Each message must have at least one valid recipient, or the message cannot be submitted for delivery.

To send a message that contains HTML, it is recommended that you provide both a plain text version of the message body and an HTML formatted version. While it is permitted to send a

message that only contains HTML, some older mail clients may not be capable of displaying the message correctly. In some cases, anti-spam software will increase the spam score of messages that do not contain a plain text message body. This can result in your message being rejected or quarantined by the mail server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SMTPSERVER](#)

SMTPMESSAGEEX Structure

This structure provides information about the contents of a message and is used by the [SubmitMessage](#) method.

```
typedef struct _SMTPMESSAGEEX
{
    DWORD    dwSize
    LPCTSTR  lpszFrom;
    LPCTSTR  lpszTo;
    LPCTSTR  lpszCc;
    LPCTSTR  lpszBcc;
    LPCTSTR  lpszDate;
    LPCTSTR  lpszSubject;
    LPCTSTR  lpszHeaders;
    LPCTSTR  lpszText;
    LPCTSTR  lpszHTML;
    LPCTSTR  lpszAttach;
    UINT     nCharSet;
    UINT     nEncType;
    DWORD    dwReserved;
} SMTPMESSAGEEX, *LPSMTPMESSAGEEX;
```

Members

dwSize

An integer value that specifies the size of the **SMTPMESSAGEEX** data structure. This must always be explicitly defined, and if the value is incorrect, an error will be returned. This structure member is used to ensure that the correct version of the structure is being passed to the method.

lpszFrom

A pointer to a string that specifies the email address of the person sending the message. This structure member must point to a valid address and cannot be NULL.

lpszTo

A pointer to a string that specifies the email addresses of one or more recipients. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member must point to at least one valid address and cannot be NULL.

lpszCc

A pointer to a string that specifies the email addresses of one or more recipients that will receive copies of the message. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member may be NULL or point to an empty string.

lpszBcc

A pointer to a string that specifies the email addresses of one or more recipients that will receive blind copies of the message. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member may be NULL or point to an empty string. Unlike the recipients specified by the *lpszTo* and *lpszCc* members, any addresses specified by this member will not be included in the header of the email message.

lpszDate

A pointer to a string that specifies the date and time for the message. This structure member may be NULL or point to an empty string. If the date is not specified, then the current date and time will be used by default. If a date is specified, it should be in the standard format as defined

by RFC822.

lpzSubject

A pointer to a string that specifies the subject of the message. This structure member may be NULL, in which case no subject will be included in the message.

lpzHeaders

A pointer to a string that specifies additional headers that should be included in the message. Header names should be separated from values by a colon, and multiple headers may be defined by separating them with a newline character. This structure member may be NULL, in which case no additional headers will be included in the message.

lpzText

A pointer to a string which contains the body of the message as plain text. Each line of text contained in the string should be terminated with a carriage-return and linefeed (CRLF) pair, which is recognized as the end-of-line. If this structure member is NULL or points to an empty string, then the *lpzHTML* member must specify the body of the message.

lpzHTML

A pointer to a string which contains the message using HTML formatting. If the *lpzText* member is not NULL, then a multipart message will be created with both plain text and HTML text as the alternative. This allows mail clients to select which message body they wish to display. If the *lpzText* member is NULL or points to an empty string, then the message will only contain HTML. Although this is supported, it is not recommended because older mail clients may be unable to display the message correctly.

lpzAttach

A pointer to a string which specifies one or more file attachments for the message. If multiple files are to be attached to the message, each file name must be separated by a semi-colon. It is recommended that you provide the complete path to the file. If this structure member is NULL or points to an empty string, the message will be created without attachments.

nCharSet

A integer value which specifies the character set to use when composing the message. A value of zero specifies that the default USASCII character set should be used. The following values may also be used:

Constant	Description
MIME_CHARSET_USASCII	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the first bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_CHARSET_ISO8859_1	An 8-bit character set for most western European languages such as English, French, Spanish and German. This character set is also commonly referred to as Latin1.
MIME_CHARSET_ISO8859_2	An 8-bit character set for most central and eastern European languages such as Czech, Hungarian, Polish and Romanian. This character set is also commonly referred to as Latin2.
MIME_CHARSET_ISO8859_5	An 8-bit character set for Cyrillic languages such as Russian, Bulgarian and Serbian.

MIME_CHARSET_ISO8859_6	An 8-bit character set for Arabic languages. Note that the application is responsible for displaying text that uses this character set. In particular, any display engine needs to be able to handle the reverse writing direction and analyze the context of the message to correctly combine the glyphs.
MIME_CHARSET_ISO8859_7	An 8-bit character set for the Greek language.
MIME_CHARSET_ISO8859_8	An 8-bit character set for the Hebrew language. Note that similar to Arabic, Hebrew uses a reverse writing direction. An application which displays this character should be capable of processing bi-directional text where a single message may include both right-to-left and left-to-right languages, such as Hebrew and English.
MIME_CHARSET_ISO8859_9	An 8-bit character set for the Turkish language. This character set is also commonly referred to as Latin5.

nEncType

A numeric identifier which specifies the encoding type to use when composing the message. A value of zero specifies that default 7bit encoding should be used. The following values may also be used:

Constant	Description
MIME_ENCODING_7BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the most significant bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_ENCODING_8BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long and all bits are used. 8-bit encoding is typically used with multibyte character sets and is the default encoding used with Unicode text.
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.

dwReserved

An unsigned integer value reserved for future use. This structure member should always have a value of zero.

Remarks

This structure is used to define the contents of a message that will be submitted for delivery using

the **SubmitMessage** method. It is required that you specify a sender, at least one recipient and a message body. Other structure members may be NULL or have a value of zero to indicate that either the value is not required, or that a default should be used. It is recommended that you initialize all of the structure members to a value of zero using the **ZeroMemory** function prior to populating the structure.

Note that you must explicitly define the size of the structure by setting the value of the **dwSize** member variable. This ensures that the correct version of the structure is being passed to the method.

Email addresses may be specified as simple addresses, or as commented addresses that include the sender's name or other information. For example, any one of these address formats are acceptable:

```
user@domain.tld
User Name <user@domain.tld>
user@domain.tld (User Name)
```

To specify multiple addresses, you should separate each address by a comma or semi-colon. Note that the **lpzFrom** member cannot specify multiple addresses, however it is permitted with the **lpzTo**, **lpzCc** and **lpzBcc** structure members. Each message must have at least one valid recipient, or the message cannot be submitted for delivery.

If you specify a message date by assigning a value to the **lpzDate** member, and it does not include any timezone information, Coordinated Universal Time (UTC) will be used by default. This is an important consideration if you provide input from a user, because in most cases they will not include the timezone and will assume the date and time they enter is for their current timezone.

If you wish to include additional headers in the message, you can specify them in a string. Each header consists of a name and value, separated by a colon (":") character. If you wish to define multiple headers, then you can separate them with a newline (e.g.: a linefeed character or combination of a carriage-return and linefeed). Extraneous leading and trailing whitespace are trimmed from header names and values. Invalid names or values will be ignored and will not generate an error.

To send a message that contains HTML, it is recommended that you provide both a plain text version of the message body and an HTML formatted version. While it is permitted to send a message that only contains HTML, some older mail clients may not be capable of displaying the message correctly. In some cases, anti-spam software will increase the spam score of messages that do not contain a plain text message body. This can result in your message being rejected or quarantined by the mail server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SMTPSERVER](#)

SMTPSERVER Structure

This structure provides connection information for a mail server and is used by the [SubmitMessage](#) method.

```
typedef struct _SMTPSERVER
{
    LPCTSTR lpszHostName;
    LPCTSTR lpszUserName;
    LPCTSTR lpszPassword;
    UINT nHostPort;
    UINT nTimeout;
    DWORD dwOptions;
    DWORD dwReserved;
} SMTPSERVER, *LPSMTPSERVER;
```

Members

lpszHostName

A pointer to a string that specifies the host name or IP address of the mail server. This structure member cannot be NULL.

lpszUserName

A pointer to a string that specifies the username that will be used to authenticate the client session. If the mail server does not require authentication, this structure member can be NULL or point to an empty string.

lpszPassword

A pointer to a string that specifies the password that will be used to authenticate the client session. If the mail server does not require authentication, this structure member can be NULL or point to an empty string.

nHostPort

An integer value that specifies the port number used to establish the connection. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 25. An alternative port is 587, which is commonly used by authenticated clients to submit messages for delivery. For implicit SSL connections, the default port number is 465.

nTimeout

An integer value that specifies the number of seconds that the client will wait for a response from the server before failing the operation. A value of zero specifies the default timeout period of 20 seconds.

dwOptions

An unsigned integer that specifies one or more options. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Constant	Description
SMTP_OPTION_NONE	No additional options are specified when establishing a connection with the server. A standard, non-secure connection will be used.
SMTP_OPTION_EXTENDED	Extended SMTP commands should be used if possible. This option enables features such as

	authentication and delivery status notification. If this option is not specified, the library will not attempt to use any extended features. This option is automatically enabled if a username and password are specified, or if the connection is established on port 587, because submitting messages for delivery using this port typically requires client authentication.
SMTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
SMTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
SMTP_OPTION_SECURE	This option specifies that a secure connection should be established with the server and requires that the server support either the SSL or TLS protocol. The client will initiate the secure session using the STARTTLS command.
SMTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using either the SSL or TLS protocol, and the secure session must be negotiated immediately after the connection has been established.

dwReserved

An unsigned integer value reserved for future use. This structure member should always have a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SMTPMESSAGE](#)

SMTPTTRANSFERSTATUS Structure

This structure is used by the [GetTransferStatus](#) method to return information about a message being submitted for delivery.

```
typedef struct _SMTPTTRANSFERSTATUS
{
    DWORD    dwBytesTotal;
    DWORD    dwBytesCopied;
    DWORD    dwBytesPerSecond;
    DWORD    dwTimeElapsed;
    DWORD    dwTimeEstimated;
} SMTPTTRANSFERSTATUS, *LPSMTPTTRANSFERSTATUS;
```

Members

dwBytesTotal

The total number of bytes that will be transferred. If the size of the message cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the transfer is completed. This is based on the average number of bytes transferred per second.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

SYSTEMTIME Structure

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include windows.h.

SocketWrench Class Library

A general purpose TCP/IP networking library for developing client and server applications.

Reference

- [Class Methods](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

Class Name	CSocketWrench
File Name	CSWSKV10.DLL
Version	10.0.1468.2518
LibID	1437629B-0693-44DE-93ED-1482DBEFE8DC
Import Library	CSWSKV10.LIB
Dependencies	None
Standards	RFC 768, RFC 791, RFC 793

Overview

At the core of all of the SocketTools networking libraries is the Windows Sockets API. This provides a low level interface for sending and receiving data over the Internet or a local intranet using the Transmission Control Protocol (TCP) and/or User Datagram Protocol (UDP). The SocketWrench class library provides a simpler interface to the Windows Sockets API, without sacrificing features or functionality. Using SocketWrench, you can easily create client and server applications while avoiding many of the mundane tasks and common problems that developers face when building Internet applications.

This class library supports secure connections using the TLS 1.2 protocol and can also be used to create secure, customized server applications. Both implicit and explicit SSL connections are supported, enabling the class to work with a wide variety of client and server applications without requiring that you use third-party libraries or Microsoft's CryptoAPI.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This class provides an implementation of a multithreaded server which should only be used with languages that support the creation of multithreaded applications. It is important that you do not link against static libraries which were not built with support for threading.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

SocketWrench Class Methods

Class	Description
CSocketWrench	Constructor which initializes the current instance of the class
~CSocketWrench	Destructor which releases resources allocated by the class
Method	Description
Abort	Abort the connection and immediately close the socket
Accept	Accept a connection request from a remote host
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified socket to another thread
Cancel	Cancel a blocking operation
CompareAddress	Compare two IP addresses to determine if they are identical
Connect	Connect to the specified server
CreateSecurityCredentials	Create a new security credentials structure
DeleteSecurityCredentials	Delete a previously created security credentials structure
DetachHandle	Detach the handle for the current instance of this class
DisableEvents	Disable asynchronous event notification
DisableSecurity	Disable secure communication with the remote host
DisableTrace	Disable logging of socket function calls to the trace log
Disconnect	Disconnect from the current server
EnableEvents	Enable asynchronous event notification
EnableSecurity	Enable secure communication with the remote host
EnableTrace	Enable logging of socket function calls to a file
EnumNetworkAddresses	Return the list of network addresses that are configured for the local host
Flush	Flush the send and receive buffers
FormatAddress	Convert an IP address in binary format into a printable string
FreezeEvents	Suspend or resume event handling by the application
GetAdapterAddress	Return the IP or MAC assigned to the specified network adapter
GetAddress	Convert an IP address string to a binary format
GetAddressFamily	Return the address family for the specified IP address
GetDefaultHostFile	Return the fully qualified path name of the host file on the local system
GetErrorString	Return a description for the specified error code
GetExternalAddress	Return the external IP address assigned to the local system
GetFirstAlias	Return the first alias for the specified host name
GetHandle	Return the client handle used by this instance of the class

GetHostAddress	Return the IP address assigned to the specified hostname
GetHostFile	Return the name of the host file
GetHostName	Return the hostname assigned to the specified IP address
GetLastError	Return the last error code
GetLocalAddress	Return the local IP address and port number for a socket
GetLocalName	Return the hostname assigned to the local system
GetNextAlias	Return the next alias for the specified host name
GetOption	Return the current socket options
GetPeerAddress	Return the IP address of the peer that the socket is connected to
GetPeerPort	Returns the remote port number used by the client to establish the connection
GetPhysicalAddress	Return the media access control (MAC) address for the primary network adapter
GetSecurityInformation	Return information about the security characteristics of a connection
GetServiceName	Return the service name associated with a specified port number
GetServicePort	Return the port number associated with a service name
GetStatus	Report what sort of socket operation is in progress
GetStreamInfo	Return information about the current stream read or write operation
GetTimeout	Return the timeout interval for blocking operations, in seconds
HostNameToUnicode	Converts the canonical form of a host name to its Unicode version
InetEventProc	Callback method that processes events generated on the socket
IsAddressNull	Determine if the specified IP address is a null address
IsAddressRoutable	Determine if the specified IP address is routable over the Internet
IsBlocking	Determine if the socket is performing a blocking operation
IsClosed	Determine if the remote host has closed its socket
IsConnected	Determine if the socket is connected to a remote host
IsInitialized	Determine if the class has been successfully initialized
IsListening	Determine if the socket is listening for a connection
IsProtocolAvailable	Determine if the specified protocol and address family are supported
IsReadable	Determine if data can read from the socket without blocking
IsUrgent	Determine if there is any out-of-band data available to be read
IsWritable	Determine if data can be written to to the socket without blocking
Listen	Listen for client connections on the specified socket
MatchHostName	Match a host name against of list of addresses including wildcards
NormalizeHostName	Return the canonical form of a host name
Peek	Read data from the socket without removing it from the socket buffer
Read	Read data from the socket
ReadLine	Read a line of data from the socket, storing it in a string buffer

ReadStream	Read a stream of data from the socket
RegisterEvent	Register an event callback function
Reject	Reject a pending client connection
SetHostFile	Specify the name of an alternate host table
SetLastError	Set the last error code
SetOption	Set one or more options for the current socket
SetTimeout	Set the interval used when waiting for a blocking operation to complete
ShowError	Display a message box with a description of the specified error
Shutdown	Disable reception or transmission of data
StoreStream	Read a stream of data from the socket and store it in a file
Write	Write data to the socket
WriteLine	Write a line of data to the socket, terminated with a carriage-return and linefeed
WriteStream	Write a stream of data to the socket

CSocketWrench::CSocketWrench Method

`CSocketWrench()`;

The **CSocketWrench** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **InetInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[~CSocketWrench](#), [IsInitialized](#)

CSocketWrench::~CSocketWrench

`~CSocketWrench();`

The **CSocketWrench** destructor releases resources allocated by the current instance of the **CSocketWrench** object. It also uninitialized the library if there are no other concurrent uses of the class.

Remarks

When a **CSocketWrench** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the connection are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

See Also

[CSocketWrench](#)

CSocketWrench::Abort Method

```
BOOL Abort();
```

Immediately close the socket without waiting for any remaining data to be written out.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Abort** method should only be used when the connection must be closed immediately before the application terminates. This method should only be used to abort client connections and should not be used with passive (listening) sockets. Server applications that need to abort an incoming client connection should use the **Reject** method.

In most cases, the application should call the **Disconnect** method to gracefully close the connection to the remote host. Aborting the connection will discard any buffered data and may cause errors or result in unpredictable behavior.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[Cancel](#), [Disconnect](#), [Reject](#)

CSocketWrench::Accept Method

```
BOOL Accept(  
    SOCKET hServer,  
    UINT nTimeout,  
    DWORD dwOptions,  
    HWND hEventWnd,  
    UINT uEventMsg  
);  
  
BOOL Accept(  
    CSocketWrench& swServer,  
    UINT nTimeout,  
    DWORD dwOptions,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **Accept** method is used to accept a connection on a listening socket.

This method is included for backwards compatibility with legacy applications. New projects should use the **CInternetServer** class to create a server application.

Parameters

hServer

Handle to the listening socket. This argument may also reference a **CSocketWrench** object which is listening for connections. In either case, the server socket must have been created by calling the **Listen** method.

nTimeout

The number of seconds that the server will wait for a client connection before failing the operation. This value is used only for blocking connections.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option is commonly used by server applications.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.
INET_OPTION_INLINE	This option controls how urgent (out-of-band) data is handled when reading data from the socket. If set, urgent data is placed in the data stream along with non-urgent data.

INET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
INET_OPTION_SECURE	This option determines if a secure connection is established with the remote host.
INET_OPTION_SECURE_FALLBACK	This option specifies the server should permit the use of less secure cipher suites for compatibility with legacy clients. If this option is specified, the server will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_FREETHREAD	This option specifies that this instance of the class may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the class instance is synchronized across multiple threads.

hEventWnd

The handle to the event notification window. This window receives messages which notify the application of various asynchronous socket events that occur. If this argument is NULL, then the connection will be blocking and no network events will be sent to the client.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the *hEventWnd* argument is NULL, this argument should be specified as WM_NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The use of Windows event notification messages has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

To prevent this method from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **Accept** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread to accept each client session.

When a connection is accepted by the server, the original listening socket continues to listen for more connections. If no event notification window is specified, then **Accept** will block until a client attempts to connect to the server or the timeout period expires.

If you specify an event notification window, then the connection will be asynchronous. When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has

occurred. The *wParam* parameter contains the socket handle. One or more of the following event identifiers may be sent:

Constant	Description
INET_EVENT_DISCONNECT	The remote host has closed the connection. The process should read any remaining data and disconnect.
INET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the process has read at least some of the data from the socket. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_WRITE	The process can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the socket is in asynchronous mode.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional connection if more than one active connection is required.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the class instance is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call methods using that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the INET_OPTION_FREETHREAD option enables any thread to call any method in that instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same instance.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Accept](#), [Connect](#), [Disconnect](#), [Listen](#), [Reject](#)

CSocketWrench::AttachHandle Method

```
VOID AttachHandle(  
    SOCKET hSocket  
);  
  
VOID AttachHandle(  
    SOCKET hSocket,  
    DWORD dwProcessId  
);
```

The **AttachHandle** method attaches the specified socket handle to the current instance of the class.

Parameters

hSocket

The socket handle that will be attached to the current instance of the class object.

dwProcessId

The process ID for the process that currently owns the socket handle. This value may be zero to specify the current process.

Return Value

None.

Remarks

This method is used to attach a socket handle created outside of the class using the SocketWrench API. Once the socket handle is attached to the class, the other class member functions may be used with that socket. If the socket was created by a third-party library or the Windows Sockets API, then the handle will be automatically inherited by the library.

If a socket handle already has been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous socket connection from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

If the *dwProcessId* parameter specifies another process, the socket will be duplicated into the current process, attached to the current thread and the original socket handle will be closed in the other process. This enables an application to effectively take control of a connection created by another process. The original socket handle must be inheritable by the by the current process and must be an actual Windows socket handle, not a pseudo-handle. This functionality is only supported on Windows NT 4.0 and later versions of the operating system with the Microsoft TCP/IP stack. Note that Layered Service Providers (LSPs) may interfere with the ability to inherit handles across processes.

If the socket was created by another process, it is initialized by the library in a blocking state, even if was originally using asynchronous socket events. If the application requires that the socket use events, it must explicitly call **EnableEvents**. A program should never try to attach to a secure connection created by another process because the attached socket will not have the security context required to encrypt and decrypt the data exchanged with the remote host.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cswsock10.h

Import Library: cswskv10.lib

See Also

[AttachThread](#), [DetachHandle](#), [GetHandle](#)

CSocketWrench::AttachThread Method

```
DWORD AttachThread(  
    DWORD dwThreadId  
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[AttachHandle](#), [Cancel](#), [Connect](#), [DetachHandle](#), [Disconnect](#), [GetHandle](#)

CSocketWrench::Cancel Method

```
BOOL Cancel(  
    SOCKET hSocket  
);  
BOOL Cancel();
```

The **Cancel** method cancels any outstanding blocking socket operation, causing the blocking method to fail. The application may then retry the operation or terminate the connection.

Parameters

hSocket

An optional parameter that specifies the handle to the socket. If this parameter is omitted, the socket handle for the current class instance will be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

This method is typically called from within an event handler to signal that the current blocking operation should stop. It may also be used to cancel a blocking operation that is occurring on another thread.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[Abort](#), [Disconnect](#), [IsBlocking](#)

CSocketWrench::CompareAddress Method

```
BOOL CompareAddress(  
    LPINTERNET_ADDRESS lpAddress1,  
    LPINTERNET_ADDRESS lpAddress2  
);  
  
BOOL CompareAddress(  
    LPCTSTR lpszAddress1,  
    LPCTSTR lpszAddress2  
);
```

Compare two IP addresses to determine if they are identical.

Parameters

lpAddress1

A pointer to an INTERNET_ADDRESS structure that contains the first IP address to be compared. An alternate version of this method accepts a string that specifies the IP address to be compared.

lpAddress2

A pointer to an INTERNET_ADDRESS structure that contains the second IP address to be compared. An alternate version of this method accepts a string that specifies the IP address to be compared.

Return Value

If the method succeeds and the two addresses are identical, the return value is non-zero. If the method fails or the two addresses are not identical, the return value is zero. If either parameter is NULL, or the address family for the two addresses are not the same, the last error code will be updated. If the addresses are valid and in the same address family, but are not identical, the last error code will be set to NO_ERROR.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[GetHostAddress](#), [GetLocalAddress](#), [GetPeerAddress](#), [INTERNET_ADDRESS](#)

CSocketWrench::Connect Method

```
BOOL Connect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nProtocol,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPCTSTR lpszLocalAddress,  
    UINT nLocalPort,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **Connect** method is used to establish a connection with a server.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used.

nProtocol

The protocol to be used when establishing the connection. This may be one of the following values:

Constant	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This is the default protocol.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. Note that UDP is unreliable in that there is no way for the sender to know that the receiver has actually received the datagram.

nTimeout

The number of seconds to wait for a response before failing the current operation.

dwOptions

An unsigned integer used to specify one or more socket options. This parameter is constructed by using the bitwise Or operator with any of the following values:

Constant	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_DONTROUTE	This option specifies default routing should not be used. This option should not be specified unless

	absolutely necessary.
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_NODELAY	This option disables the Nagle algorithm. By default, small amounts of data written to the socket are buffered, increasing efficiency and reducing network congestion. However, this buffering can negatively impact the responsiveness of certain applications. This option disables this buffering and immediately sends data packets as they are written to the socket.
INET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
INET_OPTION_SECURE	This option specifies that a secure connection should be established with the remote host. The specific version of TLS and other security related options are provided in the <i>lpCredentials</i> parameter. If the <i>lpCredentials</i> parameter is NULL, the connection will default to using TLS 1.2 and the strongest cipher suites available. Older versions of Windows prior to Windows 7 and Windows Server 2008 R2 only support TLS 1.0 and secure connections will automatically downgrade on those platforms.
INET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
INET_OPTION_FREETHREAD	This option specifies that this instance of the class may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the class

instance is synchronized across multiple threads.

lpszLocalAddress

A pointer to a null terminated string that specifies the local IP address that the socket should be bound to. If this parameter is NULL, then an appropriate address will automatically be used. A specific address should only be used if it is required by the application.

nLocalPort

The local port number that the socket should be bound to. If this parameter is set to zero, then an appropriate port number will automatically be used. A specific port number should only be used if it is required by the application.

hEventWnd

The handle to the event notification window. This window receives messages which notify the application of various asynchronous socket events that occur. If this argument is NULL, then the connection will be blocking and no network events will be sent to the client.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the *hEventWnd* argument is NULL, this argument should be specified as WM_NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The use of Windows event notification messages has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

To prevent this method from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **Connect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

If you specify an event notification window, then the connection will be asynchronous. When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the socket handle. One or more of the following event identifiers may be sent:

Constant	Description
INET_EVENT_CONNECT	The connection to the remote host has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
INET_EVENT_DISCONNECT	The remote host has closed the connection. The process should read any remaining data and disconnect.
INET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the process has read at least some

	of the data from the socket. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_WRITE	The process can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the socket is in asynchronous mode.

To cancel asynchronous notification and return the socket to a blocking mode, use the **DisableEvents** method.

It is not recommended that you disable the Nagle algorithm by specifying the INET_OPTION_NODELAY flag unless it is absolutely required. Doing so can have a significant, negative impact on the performance of the application and network.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the class instance is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call methods using that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the INET_OPTION_FREETHREAD option enables any thread to call any method in that instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same instance.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableEvents](#), [Disconnect](#), [EnableEvents](#)

CSocketWrench::CreateSecurityCredentials Method

```
BOOL CreateSecurityCredentials(  
    DWORD dwProtocol,  
    DWORD dwOptions,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName  
);
```

```
BOOL CreateSecurityCredentials(  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName  
);
```

```
BOOL CreateSecurityCredentials(  
    LPCTSTR lpszCertName  
);
```

The **CreateSecurityCredentials** method establishes the security credentials for the connection.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols.

	This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpzUserName

A pointer to a string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpzPassword

A pointer to a string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpzCertStore

A pointer to a string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.

lpzCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Example

```
pSocket->CreateSecurityCredentials(lpzCertName);  
  
bConnected = pSocket->Connect(lpzHostName,  
                              INET_PORT_HTTP,  
                              INET_PROTOCOL_TCP,  
                              INET_TIMEOUT,  
                              INET_OPTION_SECURE);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cswsock10.h

Import Library: cswskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [DeleteSecurityCredentials](#), [GetSecurityInformation](#), [SECURITYCREDENTIALS](#)

CSocketWrench::DeleteSecurityCredentials Method

```
VOID DeleteSecurityCredentials();
```

The **DeleteSecurityCredentials** method releases the security credentials for the current connection.

Parameters

None.

Return Value

None.

Remarks

This method can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated. The security credentials are released when the class destructor is called, so it is normally not required that the application explicitly call this method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateSecurityCredentials](#)

CSocketWrench::DetachHandle Method

```
SOCKET DetachHandle();
```

The **DetachHandle** method detaches the socket handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active connection, the value `INVALID_SOCKET` will be returned.

Remarks

This method is used to detach a socket handle created by the class for use with the SocketWrench API. Once the socket handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly closed at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

See Also

[AttachHandle](#), [GetHandle](#)

CSocketWrench::DisableEvents Method

BOOL DisableEvents();

Disables event notifications, preventing messages from being posted to the application's message queue.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified socket handle. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the socket has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a socket handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[EnableEvents](#), [RegisterEvent](#)

CSocketWrench::DisableSecurity Method

```
INT DisableSecurity();
```

The **DisableSecurity** method disables a secure session with the remote host.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableSecurity** method disables a secure session, with subsequent calls to **Read** and **Write** sending and receiving unencrypted data. It is important to note that because this method sends a shutdown message to terminate the secure session, this may cause connection to be closed by the remote host.

This method does not close the socket. Use the **Disconnect** method to close the socket and release the resources allocated for the current session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[CreateSecurityCredentials](#), [DeleteSecurityCredentials](#), [EnableSecurity](#)

CSocketWrench::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[EnableTrace](#)

CSocketWrench::Disconnect Method

BOOL Disconnect();

Terminate the connection, closing the socket and releasing the memory allocated for the session.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

Once the connection has been terminated, the class instance socket handle is no longer valid and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new connection is established. An application should always consider the socket handle to be opaque and never depend on it being a specific value.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[Abort](#), [Accept](#), [Connect](#), [DisableEvents](#), [EnableEvents](#), [Listen](#)

CSocketWrench::EnableEvents Method

```
BOOL EnableEvents(  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the socket notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a network event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a network event occurs. This allows an application to monitor the status of different socket operations.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Constant	Description
INET_EVENT_CONNECT	The connection to the remote host has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
INET_EVENT_DISCONNECT	The remote host has closed the connection to the client. The client should read any remaining data and disconnect.
INET_EVENT_READ	Data is available to be read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
INET_EVENT_WRITE	The application can now send data to the remote host. This notification is sent after a connection has been established, or

	after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
INET_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The application may attempt to retry the operation, or may disconnect from the remote host and report an error to the user.
INET_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[DisableEvents](#), [RegisterEvent](#)

CSocketWrench::EnableSecurity Method

```
BOOL EnableSecurity();  
BOOL EnableSecurity(  
    LPCTSTR lpszCertName  
);  
BOOL EnableSecurity(  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName  
);
```

The **EnableSecurity** method enables a secure session with the remote host.

Parameters

lpszCertStore

A pointer to a string which specifies the name of certificate store.

lpszCertName

A pointer to a string which specifies the common name for the certificate that will be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **EnableSecurity** method enables a secure communications session with the remote host, automatically negotiating the encryption algorithm and validating the certificate. If the socket was created using the **Connect** method to establish a client connection, then **EnableSecurity** will initiate the handshake with the remote host to establish a secure session. If the **Accept** method was used to accept a connection from a client, then the method will block and wait for the remote host to initiate the handshake.

This method is useful if the application needs to establish an initial, non-secure connection to the remote host and then negotiate a secure connection at a later point. If the method succeeds, all subsequent calls to **Read** and **Write** to receive and send data will be encrypted.

If no arguments are specified, then the security credentials established with a previous call to **CreateSecurityCredentials** will be used. If a certificate name is specified, then the current security credentials will be updated to use that certificate.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[CreateSecurityCredentials](#), [DeleteSecurityCredentials](#), [DisableSecurity](#)

CSocketWrench::EnableTrace Method

```
BOOL EnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **EnableTrace** method enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the remote host.

Trace method logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableTrace](#)

CSocketWrench::EnumNetworkAddresses Method

```
INT EnumNetworkAddresses(  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddressList,  
    INT nMaxAddresses  
);  
  
INT EnumNetworkAddresses(  
    LPINTERNET_ADDRESS lpAddressList,  
    INT nMaxAddresses  
);
```

The **EnumNetworkAddresses** method returns the list of network addresses that are configured for the local host.

Parameters

nAddressFamily

An integer which identifies the type of IP address that should be returned by this method. It may be one of the following values:

Constant	Description
INET_ADDRESS_ANY	Return both IPv4 or IPv6 addresses assigned to the local host, depending on how the system is configured and which network interfaces are enabled. This option is only recommended for applications that require support for IPv6 connections.
INET_ADDRESS_IPV4	Return only the IPv4 addresses assigned to the local host. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Return only the IPv6 addresses assigned to the local host. All bytes in the <i>ipNumber</i> array are significant. This option is only recommended for those applications that require support for IPv6 connections.

lpAddressList

A pointer to an array of [INTERNET_ADDRESS](#) structures that will contain the IP address of each local network interface. This parameter may be NULL, in which case the method will only return the number of available addresses.

nMaxAddresses

Maximum number of addresses to be returned. If the *lpAddressList* parameter is NULL, this value must be zero.

Return Value

If the method succeeds, the return value is the number of network addresses that are configured for the local host. If the method fails, the return value is INET_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

If the *nAddressFamily* parameter is specified as INET_ADDRESS_ANY, the application must be prepared to accept IPv6 addresses returned by this method. On Windows Vista and later versions of the operating system, IPv6 support is enabled and the local network adapter will have IPv6 addresses assigned to them by default. For legacy applications that only recognize IPv4 addresses, the *nAddressFamily* parameter should always be specified as INET_ADDRESS_IPV4 to ensure that only IPv4 addresses are returned.

This method will ignore addresses that are bound to a disabled interface, as well as those addresses bound to a virtual loopback interface. For example, although the loopback address 127.0.0.1 is a valid network address, it will not be included in list of addresses returned by this method.

The first IPv4 or IPv6 address returned by this method is typically the address assigned to the primary network adapter on the local system. However, your application should not depend on addresses being returned in any particular order. If the system has virtualization software installed, this method may also include the IP addresses assigned to any virtualized network adapters installed by that software.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[GetAdapterAddress](#), [GetHostAddress](#), [GetLocalAddress](#)

CSocketWrench::Flush Method

```
BOOL Flush();
```

The **Flush** method flushes the internal send and receive buffers used by the socket.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Flush** method will flush any data waiting to be read or written to the remote host . It is important to note that this method is not similar to flushing data to a disk file; it does not ensure that a specific block of data has been written to the socket. For example, you should never call this function immediately after calling the **Write** method or prior to calling the **Disconnect** method.

An application never needs to use the **Flush** method under normal circumstances. This method is only to be used when the application needs to immediately return the socket to an inactive state with no pending data to be read or written. Calling this function may result in data loss and should only be used if you understand the implications of discarding any data which has been sent by the remote host.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[IsReadable](#), [IsWritable](#), [Read](#), [Write](#)

CSocketWrench::FormatAddress Method

```
INT FormatAddress(  
    LPINTERNET_ADDRESS lpAddress,  
    LPTSTR lpszAddress,  
    INT cchAddress  
);  
  
INT FormatAddress(  
    LPINTERNET_ADDRESS lpAddress,  
    CString& strAddress  
);
```

The **FormatAddress** method converts a numeric IP address to a printable string. The format of the string depends on whether an IPv4 or IPv6 address is specified.

Parameters

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure which specifies the numeric IP address that should be converted to a string.

lpszAddress

A pointer to the buffer that will contain the formatted IP address. This buffer should be at least 46 characters in length. This may also reference a **CString** object which will contain the formatted address when the method returns.

cchAddress

The maximum number of characters that can be copied into the address buffer.

Return Value

If the method succeeds, the return value is the length of the IP address string. If the method fails, the return value is `INET_ERROR`, meaning that the IP address could not be converted into a string. Typically this indicates that the pointer to the `INTERNET_ADDRESS` structure is invalid, or the data does not specify a valid IP address family.

Remarks

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostAddress](#), [GetLocalAddress](#), [GetPeerAddress](#), [INTERNET_ADDRESS](#)

CSocketWrench::FreezeEvents Method

```
INT FreezeEvents(  
    BOOL bFreeze  
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the application.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all socket events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the program has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableEvents](#), [EnableEvents](#), [RegisterEvent](#)

CSocketWrench::GetAdapterAddress Method

```
INT GetAdapterAddress(  
    INT nAdapterIndex,  
    INT nAddressType,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

```
INT GetAdapterAddress(  
    INT nAdapterIndex,  
    INT nAddressType,  
    CString& strAddress  
);
```

Return the IP or MAC assigned to the specified network adapter.

Parameters

nAdapterIndex

An integer value that identifies the network adapter.

nAddressType

An integer value which specifies the type of address that should be returned:

Constant	Description
INET_ADAPTER_IPV4	The address string will contain the primary IPv4 unicast address assigned to the network adapter.
INET_ADAPTER_IPV6	The address string will contain the primary IPv6 unicast address assigned to the network adapter.
INET_ADAPTER_MAC	The address string will contain the media access control (MAC) address assigned to the network adapter.

lpszAddress

A string buffer that will contain the IP or MAC address assigned to the adapter. This parameter cannot be NULL and it is recommended that it be at least 64 characters in length to provide enough space for any address type. An alternate form of the method accepts a **CString** argument which will contain the hostname.

nMaxLength

The maximum number of characters that can be copied into the string buffer, including the terminating null character. If the buffer is too small to store the complete address, this method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied to the string buffer, not including the terminating null character. A return value of zero indicates that the requested address type has not been assigned to the adapter. If the method fails, the return value is `INET_ERROR` and this typically indicates that either the adapter index is invalid or the string buffer is not large enough to store the complete address. To get extended error information, call **GetLastError**.

Remarks

The **GetAdapterAddress** method will return the IPv4, IPv6 or MAC address assigned to a specific network adapter. The primary network adapter has an index value of zero, and it increments for each adapter that is configured on the local system.

The media access control (MAC) address is a 48 bit or 64 bit value that is assigned to each network interface and is used for identification and access control. All network devices on the same subnet must be assigned their own unique MAC address. Unlike IP addresses which may be assigned dynamically and can be frequently changed, MAC addresses are considered to be more permanent because they are usually assigned by the device manufacturer and stored in firmware. Note that in some cases it is possible to change the address assigned to a device, and virtual network interfaces may have configurable MAC addresses.

This method returns the MAC address string as sequence of hexadecimal values separated by a colon. An example of a 48 bit MAC address would be "01:23:45:67:89:AB". Note that some virtual network adapters may not have a MAC address assigned to them, in which case this method would return zero.

This method will ignore network adapters that have been disabled, as well as those that are bound to a virtual loopback interface. If the system has dial-up networking or virtualization software installed, this method may also return IP addresses assigned to a virtualized network adapters installed by that software.

Example

```
// Display the IPv4 address assigned to each network adapter
for (INT nIndex = 0;; nIndex++)
{
    CString strAddress;

    if (pSocket->GetAdapterAddress(nIndex, INET_ADAPTER_IPV4, strAddress) ==
INET_ERROR)
        break;

    _tprintf(_T("Adapter %d: %s\n"), nIndex, szAddress);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[EnumNetworkAddresses](#), [GetLocalAddress](#), [GetLocalName](#)

CSocketWrench::GetAddress Method

```
INT GetAddress(  
    LPCTSTR lpszAddress,  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddress  
);
```

```
INT GetAddress(  
    LPCTSTR lpszAddress,  
    LPINTERNET_ADDRESS lpAddress  
);
```

The **GetAddress** method converts an IP address string to binary format.

Parameters

lpszAddress

A pointer to a null terminated string which specifies an IP address. This method recognizes the format for both IPv4 and IPv6 format addresses.

nAddressFamily

An integer which identifies the type of IP address specified by the *lpszAddress* parameter. It may be one of the following values:

Constant	Description
INET_ADDRESS_UNKNOWN	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on the value of the <i>lpszAddress</i> parameter.
INET_ADDRESS_IPV4	Specifies that the address should be in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero. If the <i>lpszAddress</i> parameter does not specify a valid IPv4 address string, this method will fail.
INET_ADDRESS_IPV6	Specifies that the address should be in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. If the <i>lpszAddress</i> parameter does not specify a valid IPv6 address string, this method will fail.

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the IP address.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the *nAddressFamily* parameter is specified as INET_ADDRESS_UNKNOWN, the application must be prepared to handle IPv6 addresses because it is possible that an IPv6 address string has been specified. For legacy applications that only recognize IPv4 addresses, the *nAddressFamily* member

should always be specified as `INET_ADDRESS_IPV4` to ensure that only IPv4 addresses are returned and any attempt to specify an IPv6 address string would result in an error.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **IsProtocolAvailable** method. If an IPv6 stack is not installed, this method will fail if the *lpszAddress* parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FormatAddress](#), [IsAddressNull](#), [IsAddressRoutable](#), [IsProtocolAvailable](#), [INTERNET_ADDRESS](#)

CSocketWrench::GetAddressFamily Method

```
INT GetAddressFamily(  
    LPCTSTR lpszAddress,  
);
```

Return the address family for the specified IP address.

Parameters

lpszAddress

A pointer to a string which specifies an IP address. This method recognizes the format for both IPv4 and IPv6 format addresses.

Return Value

If the method succeeds, the return value is the address family for the specified IP address and may be one of the values listed below. If the method fails, the return value is INET_ADDRESS_UNKNOWN. To get extended error information, call the **GetLastError** method.

Constant	Description
INET_ADDRESS_IPV4	The address passed to the method is a valid IPv4 address.
INET_ADDRESS_IPV6	The address passed to the method is a valid IPv6 address.

Remarks

The **GetAddressFamily** method returns the address family associated with the specified IP address string. This can be used to determine if a string specifies a valid IPv4 or IPv6 address that can be passed to other methods such as **Connect**. Note that this method will not attempt to resolve hostnames, it will only accept IP addresses.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **IsProtocolAvailable** method. If an IPv6 stack is not installed, this method will fail if the *lpszAddress* parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IsAddressNull](#), [IsAddressRoutable](#), [IsProtocolAvailable](#), [INTERNET_ADDRESS](#)

CSocketWrench::GetDefaultHostFile Method

```
INT GetDefaultHostFile(  
    LPTSTR lpszFileName,  
    INT nMaxLength  
);  
  
INT GetDefaultHostFile(  
    CString& lpszFileName  
);
```

The **GetDefaultHostFile** method returns the fully qualified path name of the host file on the local system. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **GetHostAddress** and **GetHostNames** method. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

```
ipaddress hostname [hostalias ...]
```

For example, one typical entry maps the name "localhost" to the local loopback IP address. This would be entered as:

```
127.0.0.1 localhost
```

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

The location of the default host file depends on the operating system. For Windows 95/98 and Windows Me the file is stored in C:\Windows\hosts and for Windows NT and later versions the file is stored in C:\Windows\system32\drivers\etc\hosts. Regardless of platform, there is no filename extension and this file may or may not exist on a given system.

Parameters

lpszFileName

Pointer to a string buffer that will contain the fully qualified file name to the default host file. It is recommended that this buffer be at least MAX_PATH characters in size. This parameter may be NULL, in which case the method will return the length of the string, not including the terminating null byte.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the method succeeds, the return value is length of the string. A return value of zero indicates that the default host file could not be determined for the current platform. To get extended error information, call **GetLastError**.

Remarks

This method only returns the default location of the host file and does not determine if the file actually exists. It is not required that a host file be present on the system.

The default host file is processed before performing a nameserver lookup when resolving a hostname into an IP address, or an IP address into a hostname.

To specify an alternate local host file, use the **SetHostFile** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cswsock10.h

Import Library: csWSKV10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostAddress](#), [GetHostFile](#), [GetHostName](#), [SetHostFile](#)

CSocketWrench::GetErrorString Method

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);  
  
INT GetErrorString(  
    DWORD dwErrorCode,  
    CString& strDescription  
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLastError](#), [SetLastError](#), [ShowError](#)

CSocketWrench::GetExternalAddress Method

```
INT GetExternalAddress(  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddress,  
    INT nMaxLength  
);  
  
INT GetExternalAddress(  
    INT nAddressFamily,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);  
  
INT GetExternalAddress(  
    INT nAddressFamily,  
    CString& strAddress  
);
```

The **GetExternalAddress** method returns the external IP address for the local system.

Parameters

nAddressFamily

An integer which identifies the type of IP address that should be returned by this function. It may be one of the following values:

Constant	Description
INET_ADDRESS_IPV4	Specifies that the address should be in IPv4 format. The method will attempt to determine the external IP address using an IPv4 network connection.
INET_ADDRESS_IPV6	Specifies that the address should be in IPv6 format. The method will attempt to determine the external IP address using an IPv6 network connection and requires that the local host have an IPv6 network interface installed and enabled.

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the external IP address of the local host in binary form.

lpszAddress

A pointer to a string buffer that will contain the external IP address of the local host.

nMaxLength

The maximum length of the string that will contain the IP address when the method returns.

Return Value

In the first form of the method, if it succeeds, the return value is the IP address of the local system in numeric form. If the method fails, the return value is INET_ADDRESS_NONE. In the second form, the return value is the length of the IP address string and an error is indicated by the return value INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetExternalAddress** method returns the IP address assigned to the router that connects the

local host to the Internet. This is typically used by an application executing on a system in a local network that uses a router which performs Network Address Translation (NAT). In that network configuration, the **GetLocalAddress** method will only return the IP address for the local system on the LAN side of the network unless a connection has already been established to a remote host. The **GetExternalAddress** function can be used to determine the IP address assigned to the router on the Internet side of the connection and can be particularly useful for servers running on a system behind a NAT router.

This method requires that you have an active connection to the Internet and calling this function on a system that uses dial-up networking may cause the operating system to automatically connect to the Internet service provider. An application should always check the return value in case there is an error; never assume that the return value is always a valid address. The function may be unable to determine the external IP address for the local host for a number of reasons, particularly if the system is behind a firewall or uses a proxy server that restricts access to external sites on the Internet. If the function is able to obtain a valid external address for the local host, that address will be cached by the library for sixty minutes. Because dial-up connections typically have different IP addresses assigned to them each time the system is connected to the Internet, it is recommended that this function only be used with broadband connections where a NAT router is being used.

Calling this function may cause the current thread to block until the external IP address can be resolved and should never be used in conjunction with asynchronous socket connections. If you need to call this function in an application which uses asynchronous sockets, it is recommended that you create a new thread and call this function from within that thread.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostAddress](#), [GetLocalAddress](#), [GetPeerAddress](#)

CSocketWrench::GetFirstAlias Method

```
BOOL GetFirstAlias(  
    LPCTSTR lpszHostName,  
    LPTSTR lpszHostAlias,  
    INT nMaxLength  
);
```

```
BOOL GetFirstAlias(  
    LPCTSTR lpszHostName,  
    CString& strHostAlias  
);
```

The **GetFirstAlias** method returns the first alias for the specified host name.

Parameters

lpszHostName

A pointer to a string which specifies the host name that you wish to return aliases for. This should be complete domain name.

lpszHostAlias

A string buffer which will contain the first alias for the specified host name. This string should be at least 64 bytes in length. This argument may also reference a **CString** object which will contain the host alias when the method returns.

nMaxLength

Maximum number of characters that can be copied into the *lpszHostAlias* string buffer, including the terminating null byte.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Example

```
CSocketWrench sockClient;  
CString strHostAlias;  
BOOL bResult;  
  
m_ctlListBox.ResetContent();  
  
bResult = sockClient.GetFirstAlias(m_strHostName, strHostAlias);  
if (bResult == FALSE)  
{  
    sockClient.ShowError();  
    return;  
}  
  
while (bResult)  
{  
    m_ctlListBox.AddString(strHostAlias);  
    bResult = sockClient.GetNextAlias(strHostAlias);  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostAddress](#), [GetHostName](#), [GetNextAlias](#)

CSocketWrench::GetHandle Method

```
SOCKET GetHandle();
```

The **GetHandle** method returns the socket handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the socket handle associated with the current instance of the class object. If there is no active connection, the value `INVALID_SOCKET` will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[AttachHandle](#), [DetachHandle](#), [IsInitialized](#)

CSocketWrench::GetHostAddress Method

```
INT GetHostAddress(  
    LPCTSTR lpszHostName,  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddress  
);
```

```
INT GetHostAddress(  
    LPCTSTR lpszHostName,  
    LPINTERNET_ADDRESS lpAddress  
);
```

The **InetGetHostAddress** method resolves the specified host name into an IP address in binary format.

Parameters

lpszHostName

A pointer to the name of the host to resolve; this may be a fully-qualified domain name or an IP address. This method recognizes the format for both IPv4 and IPv6 format addresses.

nAddressFamily

An integer which identifies the type of IP address to return. It may be one of the following values:

Constant	Description
INET_ADDRESS_UNKNOWN	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on how the host name can be resolved. By default, a preference will be given for returning an IPv4 address. However, if the host only has an IPv6 address, that value will be returned.
INET_ADDRESS_IPV4	Specifies that the address should be returned in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address should be returned in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the IP address of the specified host.

Return Value

If the method succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

This method can also be used to convert an address in dot notation to a binary format. If the method must perform a DNS lookup to resolve the hostname, the calling thread will block. To ensure future compatibility with IPv6 networks, it is important that the application does not make any assumptions about the format of the address. If the function returns successfully, the *ipFamily* member of the **INTERNET_ADDRESS** structure should always be checked to determine the type of address.

The *nAddressFamily* parameter is used to specify a preference for the type of address returned, however it is possible that a host may not have an IPv4 or IPv6 address record, in which case this function will fail. Although IPv4 is still the most common address used at this time, an application should not assume that because a given host name does not have an IPv4 address, that the host name is invalid.

If the *nAddressFamily* parameter is specified as `INET_ADDRESS_UNKNOWN`, the application must be prepared to handle IPv6 addresses because it is possible for a host name to have an IPv6 address assigned to it and no IPv4 address. For legacy applications that only recognize IPv4 addresses, the *nAddressFamily* member should always be specified as `INET_ADDRESS_IPV4` to ensure that only IPv4 addresses are returned.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **IsProtocolAvailable** method. If an IPv6 stack is not installed, this method will fail if the *lpszHostName* parameter specifies a host that only has an IPv6 (AAAA) DNS record.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostName](#), [GetLocalAddress](#), [GetLocalName](#), [GetPeerAddress](#), [IsProtocolAvailable](#), [INTERNET_ADDRESS](#)

CSocketWrench::GetHostFile Method

```
INT GetHostFile(  
    LPTSTR lpszFileName,  
    INT nMaxLength  
);  
  
INT GetHostFile(  
    CString& strFileName  
);
```

The **GetHostFile** method returns the name of the host file previously set using the **SetHostFile** method. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **GetHostAddress** and **GetHostNames** method.

Parameters

lpszFileName

Pointer to a string buffer that will contain the host file name. It is recommended that this buffer be at least MAX_PATH characters in size. This parameter may be NULL, in which case the method will return the length of the string, not including the terminating null character.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the method succeeds, the return value is length of the string. A return value of zero indicates that no host file has been specified or the method was unable to determine the file name. To get extended error information, call **GetLastError**. If the last error is zero, this indicates that no host file name has been specified for the current thread. If the last error is non-zero, this indicates the reason that the method failed.

Remarks

This method only returns the name of the host file that is cached in memory for the current thread. The contents of the file on the disk may have changed after the file was loaded into memory. To reload the host file or clear the cache, call the **SetHostFile** method.

If a host file has been specified, it is processed before the default host file when resolving a hostname into an IP address, or an IP address into a hostname. If the host name or address is not found, or no host file has been specified, a nameserver lookup is performed.

The host file returned by this method may be different than the default host file for the local system. To determine the file name for the default host file, use the **GetDefaultHostFile** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetDefaultHostFile](#), [GetHostAddress](#), [GetHostName](#), [SetHostFile](#)

CSocketWrench::GetHostName Method

```
INT GetHostName(  
    LPINTERNET_ADDRESS LpAddress,  
    LPTSTR LpszHostName,  
    INT cchHostName  
);
```

```
INT GetHostName(  
    LPINTERNET_ADDRESS LpAddress,  
    CString& strHostName  
);
```

The **GetHostName** method performs a reverse lookup, returning the host name associated with a given IP address.

Parameters

LpAddress

A pointer to an [INTERNET_ADDRESS](#) structure which specifies the IP address that should be resolved into a host name.

LpszHostName

A pointer to the buffer that will contain the host name. It is recommended that this buffer be at least 256 characters in length to accommodate the longest possible fully qualified domain name.

cchHostName

The maximum number of characters that can be copied into the buffer.

Return Value

If the method succeeds, the return value is the length of the hostname. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

If the method must perform a reverse DNS lookup to resolve the IP address into a host name, the calling thread will block. This method requires that the host have a PTR record, otherwise it will fail. Because many hosts do not have a PTR record, calling this method frequently may have a negative impact on the overall performance of the application.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **IsProtocolAvailable** method. If an IPv6 stack is not installed, this method will fail if the *LpAddress* parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostAddress](#), [GetLocalAddress](#), [GetLocalName](#), [GetPeerAddress](#), [IsProtocolAvailable](#), [INTERNET_ADDRESS](#)

CSocketWrench::GetLastError Method

```
DWORD GetLastError();  
  
DWORD GetLastError(  
    CString& strDescription  
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError(0)** when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SOCKET or INET_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[GetErrorString](#), [SetLastError](#), [ShowError](#)

CSocketWrench::GetLocalAddress Method

```
INT GetLocalAddress(  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddress,  
    UINT * lpnPort  
);
```

```
INT GetLocalAddress(  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

```
INT GetLocalAddress(  
    CString& strAddress,  
    UINT * lpnPort  
);
```

The **GetLocalAddress** method returns the local IP address and port number for the current socket.

Parameters

nAddressFamily

An integer which identifies the type of IP address to return. It may be one of the following values:

Constant	Description
INET_ADDRESS_UNKNOWN	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on the type of connection that was established. If the <i>hSocket</i> parameter is INVALID_SOCKET, a preference will be given for returning an IPv4 address. However, if the local host only has an IPv6 address, that value will be returned.
INET_ADDRESS_IPV4	Specifies that the address should be returned in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address should be returned in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the IP address of the local host. If there is no active connection, this function will attempt to determine the IP address of the local host assigned by the system. If the address is not required, this parameter may be NULL.

lpszAddress

A pointer to a null terminated string that will contain the IP address of the local host. If this

version of the method is used, the IP address is converted to a string format using the **FormatAddress** method. The string should be able to store at least 46 characters to ensure that both IPv4 and IPv6 formatted addresses can be returned without the possibility of a buffer overrun. An alternate form of the method accepts a **CString** argument which will contain the local address.

lpnPort

A pointer to an unsigned integer that will contain the local port number. If there is an active connection, this parameter will be set to the local port that the socket is bound to. If there is no active connection, this parameter is ignored. If the local port number is not required, this parameter may be NULL.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

To ensure future compatibility with IPv6 networks, it is important that the application does not make any assumptions about the format of the address. If the function returns successfully, the *ipFamily* member of the **INTERNET_ADDRESS** structure should always be checked to determine the type of address.

If the *nAddressFamily* parameter is specified as `INET_ADDRESS_UNKNOWN`, the application must be prepared to handle IPv6 addresses because it is possible for the local host to have an IPv6 address assigned to it and no IPv4 address. For legacy applications that only recognize IPv4 addresses, the *nAddressFamily* member should always be specified as `INET_ADDRESS_IPV4` to ensure that only IPv4 addresses are returned.

If the system is connected to the Internet through a local network and/or uses a router that performs Network Address Translation (NAT), the **GetLocalAddress** method will return the local, non-routable IP address assigned to the local system. To determine the public IP address has been assigned to the system, you should use the **GetExternalAddress** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetExternalAddress](#), [GetHostAddress](#), [GetHostName](#), [GetLocalName](#), [GetPeerAddress](#), [INTERNET_ADDRESS](#)

CSocketWrench::GetLocalName Method

```
INT GetLocalName(  
    LPTSTR lpszHostName,  
    INT cchHostName  
);  
  
INT GetLocalName(  
    CString& strHostName  
);
```

The **GetLocalName** method returns the hostname assigned to the local system.

Parameters

lpszHostName

A pointer to the buffer that will contain the hostname. This parameter cannot be NULL. An alternate form of the method accepts a **CString** argument which will contain the local hostname.

cchHostName

The maximum number of characters that can be copied into the address buffer.

Return Value

If the method succeeds, the return value is the length of the hostname. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostAddress](#), [GetHostName](#), [GetLocalAddress](#), [GetPeerAddress](#)

CSocketWrench::GetNextAlias Method

```
BOOL GetNextAlias(  
    LPTSTR lpszHostAlias,  
    INT nMaxLength  
);  
  
BOOL GetNextAlias(  
    CString& strHostAlias  
);
```

The **GetNextAlias** method returns the next alias for the host name specified in the call to **GetFirstAlias**.

Parameters

lpszHostAlias

A string buffer which will contain the next alias for the specified host name. This string should be at least 64 bytes in length. This argument may also reference a **CString** object which will contain the host alias when the method returns.

nMaxLength

Maximum number of characters that can be copied into the *lpszHostAlias* string buffer, including the terminating null byte.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Example

```
CSocketWrench sockClient;  
CString strHostAlias;  
BOOL bResult;  
  
m_ctlListBox.ResetContent();  
  
bResult = sockClient.GetFirstAlias(m_strHostName, strHostAlias);  
if (bResult == FALSE)  
{  
    sockClient.ShowError();  
    return;  
}  
  
while (bResult)  
{  
    m_ctlListBox.AddString(strHostAlias);  
    bResult = sockClient.GetNextAlias(strHostAlias);  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cswsock10.h

Import Library: cswskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFirstAlias](#), [GetHostAddress](#), [GetHostName](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CSocketWrench::GetOption Method

```
INT GetOption(  
    DWORD dwOption,  
    LPBOOL lpbEnabled  
);
```

The **GetOption** method is used to determine if a specific socket option has been enabled.

Parameters

dwOption

An unsigned integer used to specify one of the socket options. These options cannot be combined. The following values are recognized:

Constant	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_KEEPAIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option is commonly used by server applications.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.

lpbEnabled

A pointer to a boolean flag. If the option is enabled, the flag is set to a non-zero value, otherwise it is set to a value of zero.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[Connect](#), [SetOption](#)

CSocketWrench::GetPeerAddress Method

```
INT GetPeerAddress(  
    LPINTERNET_ADDRESS LpAddress,  
    UINT * LpnRemotePort  
);  
  
INT GetPeerAddress(  
    LPTSTR LpszAddress,  
    INT nMaxLength  
);  
  
INT GetPeerAddress(  
    CString& strAddress  
);
```

The **GetPeerAddress** method returns the peer IP address and remote port number for the specified socket.

Parameters

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the IP address of the remote host that is connected to the socket.

lpnRemotePort

A pointer to an unsigned integer that will contain the port number of the remote host that is connected to the socket.

lpszAddress

A pointer to a string buffer that will contain the formatted IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length.

nMaxLength

The maximum number of characters that can be copied into the address buffer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

If this method is called by a server application in response to a `INET_EVENT_ACCEPT` event, it will return the IP address and port number for the client that is attempting to establish the connection. If the peer address is unavailable, the *ipFamily* member of the `INTERNET_ADDRESS` structure will be zero.

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

It is not recommended that you use the port number for anything other than informational and logging purposes. Server applications should not make any assumptions about the specific port number or range of port numbers that a client is using when establishing a connection to the

server. The ephemeral port number that a client is bound to can vary based on the client operating system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[GetHostAddress](#), [GetHostName](#), [GetLocalAddress](#), [GetLocalName](#), [GetPeerPort](#), [INTERNET_ADDRESS](#)

CSocketWrench::GetPhysicalAddress Method

```
BOOL GetPhysicalAddress(  
    LPTSTR lpszAddress,  
    UINT cchAddress  
);  
  
BOOL GetPhysicalAddress(  
    CString& strAddress  
);
```

Return the media access control (MAC) address for the primary network adapter.

Parameters

lpszAddress

A string buffer that will contain the address in a printable format when the function returns. This parameter cannot be NULL. An alternate form of the method accepts a **CString** argument which will contain the address.

cchAddress

The maximum number of characters that can be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetPhysicalAddress** method returns the media access control (MAC) address for the primary network adapter. This is a 48 bit or 64 bit address that is assigned to each network interface and is used for identification and access control. All network devices on the same subnet must be assigned their own unique MAC address. Unlike IP addresses which may be assigned dynamically and can be frequently changed, MAC addresses are considered to be more permanent because they are usually assigned by the device manufacturer and stored in firmware. Note that in some cases it is possible to change the address assigned to a device, and virtual network interfaces may have configurable MAC addresses.

This method returns the MAC address as a printable string, with each byte of the address as a two-digit hexadecimal value separated by a colon. The string buffer passed to the method should be at least 20 characters long to accommodate the address and terminating null character. An example of a 48 bit address would be "01:23:45:67:89:AB". If the local system is multi-homed (having more than one network adapter) then this method will return the MAC address for the primary network adapter.

This method is provided for backwards compatibility with previous versions of the library and it is recommended that new applications use the **GetAdapterAddress** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnumNetworkAddresses](#), [GetAdapterAddress](#), [GetLocalName](#), [GetHostAddress](#)

CSocketWrench::GetSecurityInformation Method

```
BOOL GetSecurityInformation(  
    LPSECURITYINFO LpSecurityInfo  
);
```

The **GetSecurityInformation** method returns security protocol, encryption and certificate information about the current client connection.

Parameters

LpSecurityInfo

A pointer to a [SECURITYINFO](#) structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;  
securityInfo.dwSize = sizeof(SECURITYINFO);  
  
if (pSocket->GetSecurityInformation(&securityInfo))  
{  
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)  
    {  
        MessageBox(NULL, _T("The connection is not secure"),  
                    _T("Connection"), MB_OK);  
    }  
    else  
    {  
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)  
        {  
            MessageBox(NULL, _T("The connection is secure"),  
                        _T("Connection"), MB_OK);  
        }  
        else  
        {  
            MessageBox(NULL, _T("The server certificate not valid"),  
                        _T("Connection"), MB_OK);  
        }  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: csuskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [CreateSecurityCredentials](#), [EnableSecurity](#), [SECURITYINFO](#)

CSocketWrench::GetServiceName Method

```
BOOL GetServiceName(  
    UINT nServicePort,  
    LPTSTR lpszServiceName,  
    INT nMaxLength  
);  
  
BOOL GetServiceName(  
    UINT nServicePort,  
    CString& strServiceName  
);
```

The **GetServiceName** method returns the service name associated with a specified port number.

Parameters

nServicePort

Port number associated with some network service.

lpszServiceName

A pointer to a string buffer that will contain the service name when the method returns. This may also reference a **CString** object that will contain the service name.

nMaxLength

An integer value which specifies the maximum number of characters that can be copied into the string buffer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetServicePort](#)

CSocketWrench::GetServicePort Method

```
UINT GetServicePort(  
    LPCTSTR lpszServiceName  
);
```

The **GetServicePort** method returns the port number associated with a service name.

Parameters

lpszServiceName

A pointer to a string which specifies the name of the service to return the port number for.

Return Value

If the method succeeds, the return value is the port number associated with a service name. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetServiceName](#)

CSocketWrench::GetStatus Method

```
INT GetStatus();
```

The **GetStatus** method returns the current status of the socket.

Parameters

None.

Return Value

If the method succeeds, the return value is the client status code. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The return value is one of the following values:

Value	Constant	Description
0	<code>INET_STATUS_UNUSED</code>	No connection has been established.
1	<code>INET_STATUS_IDLE</code>	The socket is idle and not in a blocked state
2	<code>INET_STATUS_LISTEN</code>	The socket is listening for inbound connections from a client
3	<code>INET_STATUS_CONNECT</code>	The socket is establishing a connection with a server
4	<code>INET_STATUS_ACCEPT</code>	The socket is accepting a connection from a client
5	<code>INET_STATUS_READ</code>	Data is being read from the socket
6	<code>INET_STATUS_WRITE</code>	Data is being written to the socket
7	<code>INET_STATUS_FLUSH</code>	The socket is being flushed; all data in the receive buffers is being discarded
8	<code>INET_STATUS_DISCONNECT</code>	The socket is disconnecting from the remote host

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified socket.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[IsConnected](#), [IsInitialized](#), [IsListening](#), [IsReadable](#), [IsWritable](#)

CSocketWrench::GetStreamInfo Method

```
BOOL GetStreamInfo(  
    LPINETSTREAMINFO LpStreamInfo  
);
```

The **GetStreamInfo** function fills a structure with information about the current stream I/O operation.

Parameters

lpSecurityInfo

A pointer to an **INETSTREAMINFO** structure which contains information about the status of the current operation.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetStreamInfo** method returns information about the current streaming socket operation, including the average number of bytes transferred per second and the estimated amount of time until the operation completes. If there is no operation currently in progress, this method will return the status of the last successful streaming read or write performed by the client.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified socket.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[ReadStream](#), [StoreStream](#), [WriteStream](#), [INETSTREAMINFO](#)

CSocketWrench::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the remote host. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

See Also

[Connect](#), [IsReadable](#), [IsWritable](#), [Read](#), [SetTimeout](#), [Write](#)

CSocketWrench::HostNameToUnicode Method

```
INT HostNameToUnicode(  
    LPCTSTR lpszHostName,  
    LPTSTR lpszUnicodeName,  
    INT nMaxLength  
);  
  
INT HostNameToUnicode(  
    LPCTSTR lpszHostName,  
    CString& strUnicodeName  
);
```

The **HostNameToUnicode** method converts the canonical form of a host name to its Unicode version.

Parameters

lpszHostName

Pointer to the host name as a null-terminated string. This parameter cannot be a NULL pointer or a zero length string.

lpszUnicodeName

Pointer to the string buffer that will contain the original Unicode version of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer. An alternate version of this method accepts a reference to a CString object.

nMaxLength

The maximum number of characters that can be copied to the *lpszUnicodeName* string buffer. This parameter cannot be zero, and must include the terminating null character.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **HostNameToUnicode** method will convert the encoded ASCII version of a host name to its Unicode version. Although any valid host name is accepted by this method, it is intended to convert a Punycode encoded host name to its original Unicode character encoding.

If the application is compiled using the Unicode character set, the value returned in *lpszUnicodeName* will be a Unicode string using UTF-16 encoding. If the ANSI character set is used, the value returned will be a Unicode string using UTF-8 encoding. To display a UTF-8 encoded host name, your application will need to convert it to UTF-16 using the **MultiByteToWideChar** function.

Although this method performs checks to ensure that the *lpszHostName* parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the existence of the domain name. To check if the host name exists and has a valid IP address, use the [GetHostAddress](#) method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cswsock10.h

Import Library: cswskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostAddress](#), [NormalizeHostName](#)

CSocketWrench::IsAddressNull Method

```
BOOL IsAddressNull(  
    LPCTSTR lpszAddress  
);  
  
BOOL IsAddressNull(  
    LPINTERNET_ADDRESS lpAddress  
);
```

The **IsAddressNull** method determines if the IP address is null.

Parameters

lpszAddress

A string that specifies the IP address.

lpAddress

A pointer to an INTERNET_ADDRESS structure that specifies the IP address.

Return Value

If the method succeeds and the IP address is null, or the parameter is a NULL pointer, the return value is non-zero. If the method fails or the address is not null, the return value is zero. If the address family is not supported, the last error code will be updated. If the address is valid but not null, the last error code will be set to NO_ERROR.

Remarks

A null IP address is one where all bits for the address (32 bits for IPv4 or 128 bits for IPv6) are zero. This is a special address that is typically used when creating a passive socket that should listen for connections on all available network interfaces.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: csuskv10.lib

See Also

[GetAddress](#), [IsAddressRoutable](#), [INTERNET_ADDRESS](#)

CSocketWrench::IsAddressRoutable Method

```
BOOL IsAddressRoutable(  
    LPCTSTR lpszAddress  
);  
  
BOOL IsAddressRoutable(  
    LPINTERNET_ADDRESS lpAddress  
);
```

The **IsAddressRoutable** method determines if the IP address is routable over the Internet.

Parameters

lpszAddress

A string that specifies the IP address.

lpAddress

A pointer to an INTERNET_ADDRESS structure that specifies the IP address.

Return Value

If the method succeeds and the IP address is routable over the Internet, the return value is non-zero. If the method fails or the address is not routable, the return value is zero. If the parameter is NULL, or the address family is not supported, the last error code will be updated. If the address is valid but not routable, the last error code will be set to NO_ERROR.

Remarks

A routable IP address is one that can be reached by anyone over the public Internet. These are also commonly referred to as "public addresses" which are typically assigned to networks and individual hosts by an Internet service provider. There are also certain addresses that are not routable over the Internet, and used to address systems over a local network or private intranet. This function can be used to determine if a given IP address is public (routable) or private.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[GetAddress](#), [GetExternalAddress](#), [IsAddressNull](#), [INTERNET_ADDRESS](#)

CSocketWrench::IsBlocking Method

```
BOOL IsBlocking();
```

The **IsBlocking** method is used to determine if the socket is performing a blocking operation.

Parameters

None.

Return Value

If the socket is currently performing a blocking operation, the method returns a non-zero value. If the socket is not performing a blocking operation, or the socket handle is invalid, the method returns zero.

Remarks

This method is typically used to determine if a socket that is being used by another thread is currently blocked. A socket may block when waiting to receive data from a remote host or while data is actively being exchanged. Because there can only be one blocking socket operation per thread, this method can be used to determine if a method such as **Read** or **Write** would fail because another thread is currently sending or receiving data on that socket.

It is important to note that if this method returns a non-zero value, it does not guarantee that a subsequent read or write on the socket will succeed. The application should always check the return value from methods such as **Read** and **Write** to ensure they were successful.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[IsConnected](#), [IsReadable](#), [IsWritable](#), [Read](#), [ReadLine](#), [Write](#), [WriteLine](#)

CSocketWrench::IsClosed Method

```
BOOL IsClosed();
```

The **IsClosed** method is used to determine if the remote host has closed its socket.

Parameters

None.

Return Value

If the remote host has closed its socket, the method returns a non-zero value. If the remote host has not closed its connection, or the socket handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[IsConnected](#), [IsListening](#), [IsReadable](#), [IsWritable](#)

CSocketWrench::IsConnected Method

BOOL IsConnected();

The **IsConnected** method is used to determine if the socket is currently connected to a remote host.

Parameters

None.

Return Value

If the client is connected to a remote host, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#)

CSocketWrench::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[CSocketWrench](#), [IsBlocking](#), [IsConnected](#)

CSocketWrench::IsListening Method

```
BOOL IsListening();
```

The **IsListening** method determines if the socket is listening for connection requests.

Parameters

None.

Return Value

If the socket is being used to listen for connection requests, the method returns a non-zero value. If the socket is not listening or the socket handle is invalid, the method returns zero.

Remarks

The **IsListening** method determines if the socket is being used in a server application to actively listen for incoming connection requests from client applications. A listening socket can be created using the **Listen** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[IsReadable](#), [IsWritable](#), [IsConnected](#), [Listen](#)

CSocketWrench::IsProtocolAvailable Method

```
BOOL IsProtocolAvailable(  
    INT nAddressFamily,  
    INT nProtocol  
);
```

The **IsProtocolAvailable** method determines if the operating system supports creating a socket for the specified address family and protocol.

Parameters

nAddressFamily

An integer which identifies the address family that should be checked. It should be one of the following values:

Constant	Description
INET_ADDRESS_IPV4	Specifies that the function should determine if it can create an Internet Protocol version 4 (IPv4) socket. This requires that the system have an IPv4 TCP/IP stack bound to at least one network adapter on the local system. All Windows systems include support for IPv4 by default.
INET_ADDRESS_IPV6	Specifies that the function should determine if it can create an Internet Protocol version 6 (IPv6) socket. This requires that the system have an IPv6 TCP/IP stack bound to at least one network adapter on the local system. Windows XP and Windows Server 2003 includes support for IPv6, however it is not installed by default. Windows Vista and later versions include support for IPv6 and enable it by default.

nProtocol

An integer which identifies the protocol that should be checked. It should be one of the following values:

Constant	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This requires that the system be capable of creating a stream socket using the specified address family.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. This requires that the system be capable of creating a datagram socket using the specified address family.

Return Value

If the the system is capable of creating a socket using the specified address family and protocol, this method will return a non-zero value. If the combination of address family and protocol is not supported, this method will return a value of zero.

Remarks

The **IsProtocolAvailable** method is used to determine if the operating system supports creating a particular type of socket. Typically it is used by an application to determine if the system has an IPv6 TCP/IP stack installed and configured. By default, all Windows systems will have an IPv4 stack installed if the system has a network adapter. However, not all systems may have an IPv6 stack installed, particularly older Windows XP and Windows Server 2003 systems. Note that if an IPv6 stack is not installed, the library will not recognize IPv6 addresses and cannot resolve host names that only have an IPv6 (AAAA) record, even if the address or host name is valid.

Example

```
if (!pSocket->IsProtocolAvailable(INET_ADDRESS_IPV6, INET_PROTOCOL_TCP))
{
    AfxMessageBox(_T("This system does not support IPv6"), MB_ICONEXCLAMATION);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[GetAddress](#), [GetHostAddress](#), [GetHostName](#)

CSocketWrench::IsReadable Method

```
BOOL IsReadable(  
    INT nTimeout,  
    LPDWORD lpdwAvail  
);
```

The **IsReadable** method is used to determine if data is available to be read from the remote host.

Parameters

nTimeout

Timeout for remote host response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the current thread can read data from the socket without blocking, the method returns a non-zero value. If the current thread cannot read any data without blocking, the function returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

If the connection is secure, the value returned in *lpdwAvail* will reflect the number of bytes available in the encrypted data stream. The actual amount of data available to the application after it has been decrypted will vary.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

See Also

[IsWritable](#), [Peek](#), [Read](#), [ReadLine](#), [ReadStream](#)

CSocketWrench::IsUrgent Method

BOOL IsUrgent();

The **IsUrgent** method determines if there is any out-of-band (OOB) data available to be read.

Parameters

None.

Return Value

If there is out-of-band data, the return value is non-zero. If there is no out-of-band data, or an error occurs the return value is zero. To determine if an error has occurred, use the **GetLastError** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: csuskv10.lib

See Also

[SetOption \(INET_OPTION_INLINE\)](#)

CSocketWrench::IsWritable Method

```
BOOL IsWritable(  
    INT nTimeout  
);
```

The **IsWritable** method is used to determine if data can be written to the remote host.

Parameters

nTimeout

Timeout for remote host response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If data can be written to the socket within the specified timeout period, the method returns a non-zero value. The method will return zero if the socket send buffer is full.

Remarks

The **IsWritable** method cannot be used to determine the amount of data that can be sent to the remote host without blocking the current thread. A non-zero return value only indicates that the send buffer is not full and can accept some data. In most cases, it is recommended that larger blocks of data be broken into smaller logical blocks rather than attempting to send it all of the data at once. For very large streams of data, it is recommended that you use the **WriteStream** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[IsReadable](#), [Write](#), [WriteLine](#), [WriteStream](#)

CSocketWrench::Listen Method

```
SOCKET Listen(  
    LPCTSTR lpszLocalAddress,  
    UINT nLocalPort,  
    UINT nBackLog,  
    DWORD dwOptions,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **Listen** method creates a listening socket and specifies the maximum number of connection requests that will be queued.

This method is included for backwards compatibility with legacy applications. New projects should use the **CInternetServer** class to create a server application.

Parameters

lpszLocalAddress

A pointer to a string which specifies the local IP address that the socket should be bound to. If this parameter is NULL or points to an empty string, a client may establish a connection using any valid network interface configured on the local system. If an address is specified, then a client may only establish a connection with the system using that address.

nLocalPort

The local port number that the socket should be bound to. This value must be greater than zero. Port numbers less than 1024 are considered reserved ports and may require that the process execute with administrative privileges and/or require changes to the default firewall rules to permit inbound connections.

nBacklog

The maximum length of queue of pending connections. On Windows workstations, the maximum backlog value is 5. On Windows servers, the maximum value is 200.

dwOptions

An unsigned integer used to specify one or more socket options. The following values are recognized:

Constant	Description
INET_OPTION_NONE	No option specified. If the address and port number are in use by another application or a closed socket which was listening on this port is still in the TIME_WAIT state, the function will fail.
INET_OPTION_REUSEADDRESS	This option enables a server application to listen for connections using the specified address and port number even if they were in use recently. This is typically used to enable an application to close the listening socket and immediately reopen it without getting an error that the address is in use.
INET_OPTION_EXCLUSIVE	This option specifies the local address and port number is for the exclusive use by the current process, preventing another application from forcibly

	binding to the same address. If another process has already bound a socket to the address provided by the caller, this function will fail.
--	--

hEventWnd

The handle to the event notification window. This window receives messages which notify the application of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the method succeeds, the return value is a handle to the socket. If the method fails, the return value is INVALID_SOCKET. To get extended error information, call **GetLastError**.

Remarks

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

If the INET_OPTION_REUSEADDRESS option is not specified, an error may be returned if a listening socket was recently created for the same local address and port number. By default, once a listening socket is closed there is a period of time that all applications must wait before the address can be reused (this is called the TIME_WAIT state). The actual amount of time depends on the operating system and configuration parameters, but is typically two to four minutes. Specifying this option enables an application to immediately re-use a local address and port number that was previously in use. Note that this does not permit more than one server to bind to the same address.

If the INET_OPTION_EXCLUSIVE option is specified, the local address and port number cannot be used by another process until the listening socket is closed. This can prevent another application from forcibly binding to the same listening address as your server. This option can be useful in determining whether or not another process is already bound to the address you wish to use, but it may also prevent your server application from restarting immediately, regardless if the INET_OPTION_REUSEADDRESS option has also been specified.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the method will fail.

The use of Windows event notification messages has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

To prevent this method from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **Listen** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread to accept each client session.

When a message is posted to the notification window, the low word of the *lParam* parameter

contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the socket handle. One or more of the following event identifiers may be sent:

Constant	Description
INET_EVENT_ACCEPT	An incoming client connection is pending. The connection will be assigned to a new socket. This event is only generated if the socket is in asynchronous mode.

To cancel asynchronous notification and return the socket to a blocking mode, use the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cswsock10.h

Import Library: cswskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Accept](#), [DisableEvents](#), [EnableEvents](#), [Reject](#)

CSocketWrench::MatchHostName Method

```
BOOL MatchHostName(  
    LPCTSTR lpszHostName,  
    LPCTSTR lpszHostMask  
    BOOL bResolve  
);
```

The **MatchHostName** method matches a host name against one or more strings that may contain wildcards.

Parameters

lpszHostName

A pointer to a string which specifies the host name or IP address to match.

lpszHostMask

A pointer to a string which specifies one or more values to match against the host name. The asterisk character can be used to match any number of characters in the host name, and the question mark can be used to match any single character. Multiple values may be specified by separating them with a semicolon.

bResolve

A boolean value which specifies if the host name or IP address should be resolved when matching the host against the mask string. If this parameter is non-zero, two checks against the host mask string will be performed; once for the host name specified and once for its IP address. If this parameter is zero, then the match is made only against the host name string provided.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **MatchHostName** method provides a convenient way for an application to determine if a given host name matches one or more mask strings which may contain wildcard characters. For example, the host name could be "www.microsoft.com" and the host mask string could be "*.microsoft.com". In this example, the method would return a non-zero value indicating the host name matched the mask. However, if the mask string was "*.net" then the method would return zero, indicating that there was no match. Multiple mask values can be combined by separating them with a semicolon; for example, the mask "*.com;*.org" would match any host name in either the .com or .org top-level domains.

If an internationalized domain name (IDN) is specified, it will be converted internally to an ASCII string using Punycode encoding. The host mask will be matched against this encoded version of the host name, not its Unicode version.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetAddress](#), [GetHostAddress](#), [GetHostName](#), [GetLocalAddress](#), [GetPeerAddress](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CSocketWrench::NormalizeHostName Method

```
INT NormalizeHostName(  
    LPCTSTR lpszHostName,  
    LPTSTR lpszNormalized,  
    INT nMaxLength  
);  
  
INT NormalizeHostName(  
    LPCTSTR lpszHostName,  
    CString& strNormalized  
);
```

The **NormalizeHostName** method returns the canonical form of a host name in the specified buffer.

Parameters

lpszHostName

Pointer to the host name as a null-terminated string. This parameter cannot be a NULL pointer or a zero length string.

lpszNormalized

Pointer to the string buffer that will contain the canonical form of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer. An alternate version of this method accepts a reference to a CString object.

nMaxLength

The maximum number of characters that can be copied to the *lpszNormalized* string buffer. This parameter cannot be zero, and must include the terminating null character.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **NormalizeHostName** method will remove all leading and trailing whitespace characters from the host name and fold all upper-case characters to lower-case. If an internationalized domain name (IDN) containing Unicode characters is passed to this method, it will be converted to an ASCII compatible format for domain names.

If the application is compiled using the Unicode character set, the host name will be converted from UTF-16 to UTF-8 and then processed. If you are unsure if an internationalized domain name will be specified as the host name, it is recommended that you use Unicode.

Although this method performs checks to ensure that the *lpszHostName* parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the existence of the domain name. To check if the host name exists and has a valid IP address, use the [GetHostAddress](#) method.

It is recommended that you use this method if your application needs to store the host name, and if accepts a host name as user input. It is not necessary to call this method prior to calling the other methods that accept a host name as a parameter. They already normalize the host name

and perform checks to ensure it is in the correct format.

If the *lpzHostName* parameter specifies a valid IPv4 or IPv6 address string instead of a host name, this method will return a copy of that IP address in the buffer provided by the caller. This allows the method to be used in cases where a user may input either a host name or IP address. To determine if the IP address has a corresponding host name, use the [GetHostName](#) method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cswsock10.h

Import Library: cswskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetHostAddress](#), [GetHostName](#), [HostNameToUnicode](#)

CSocketWrench::Peek Method

```
INT Peek(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **Peek** method reads the specified number of bytes from the socket and copies them into the buffer, but it does not remove the data from the internal socket buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be stored. This argument may be NULL, in which case no data is copied from the socket buffers, however the function will return the number of bytes available to read.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. If the *lpBuffer* parameter is not NULL, this value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes available to read from the socket. A return value of zero indicates that there is no data available to read at that time. If the function fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **Peek** method returns data that is available to read from the socket, up to the number of bytes specified. The data returned by this method is not removed from the socket buffers. It must be consumed by a subsequent call to the **Read** method. The return value indicates the number of bytes that can be read in a single operation, up to the specified buffer size. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

If no data is available to be read, the method will return a value of zero. Using this method in a loop to poll a non-blocking socket may cause the application to become non-responsive. To determine if there is data available to be read, use the **IsReadable** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[EnableEvents](#), [IsBlocking](#), [IsReadable](#), [IsWritable](#), [Read](#), [ReadLine](#), [RegisterEvent](#), [Write](#), [WriteLine](#)

CSocketWrench::Read Method

```
INT Read(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Read(  
    LPBYTE lpBuffer,  
    INT cbBuffer,  
    LPINTERNET_ADDRESS lpAddress,  
    UINT* lpnRemotePort  
);
```

The **Read** method reads the specified number of bytes from the socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be stored.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

lpAddress

Pointer to an [INTERNET_ADDRESS](#) structure that which will contain the IP address of the remote host that sent the data being read. If this information is not required, the parameter may be specified as NULL.

lpnRemotePort

Pointer to an unsigned integer which will contain the remote port number. If this information is not required, the parameter may be specified as NULL.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the remote host has closed the connection and there is no more data available to be read. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **Read** method will read up to the specified number of bytes and store the data in the buffer provided by the caller. If there is no data available to be read at the time this method is invoked, the session thread will block until at least one byte of data becomes available, the timeout period elapses or an error occurs. This method will return if any amount of data is sent by the remote host, and will not block until the entire buffer has been filled. To avoid blocking the current thread, either create an asynchronous socket or use the **IsReadable** method to determine if there is data available to be read prior to calling this function.

The application should never make an assumption about the amount of data that will be available to read. TCP considers all data to be an arbitrary stream of bytes and does not impose any structure on the data itself. For example, if the remote host is sending data to the server in fixed 512 byte blocks of data, it is possible that a single call to the **Read** method will return only a partial block of data, or it may return multiple blocks combined together. It is the responsibility of the

application to buffer and process this data appropriately.

For applications that are built using the Unicode character set, it is important to note that the buffer is an array of bytes, not characters. If the remote host is sending string data to the server, it must be read as a stream of bytes and converted using the **MultiByteToWideChar** function. If the remote host is sending lines of text terminated with a linefeed or carriage return and linefeed pair, the **ReadLine** method will return a line of text at a time and perform this conversion for you.

When **Read** is called and the socket is in non-blocking mode, it is possible that the method will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cswsock10.h

Import Library: csWSKV10.lib

See Also

[EnableEvents](#), [IsBlocking](#), [IsReadable](#), [IsWritable](#), [Peek](#), [ReadLine](#), [RegisterEvent](#), [Write](#), [WriteLine](#)

CSocketWrench::ReadLine Method

```
BOOL ReadLine(  
    LPTSTR lpszBuffer,  
    LPINT lpnLength  
);  
  
BOOL ReadLine(  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);  
  
BOOL ReadLine(  
    CString& strBuffer,  
    INT nMaxLength  
);
```

The **ReadLine** method reads up to a line of data from the socket and returns it in a string buffer.

Parameters

lpszBuffer

Pointer to the string buffer that will contain the data when the method returns. The string will be terminated with a null character, and will not contain the end-of-line characters. An alternate form of the method accepts a **CString** argument which will contain the line of text returned by the server.

lpnLength

A pointer to an integer value which specifies the length of the buffer. The value should be initialized to the maximum number of characters that can be copied into the string buffer, including the terminating null character. When the method returns, its value will updated with the actual length of the string.

nMaxLength

An integer value which specifies the maximum length of the buffer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ReadLine** method reads data from the socket and copies into a specified string buffer. Unlike the **Read** method which reads arbitrary bytes of data, this method is specifically designed to return a single line of text data in a string. When an end-of-line character sequence is encountered, the method will stop and return the data up to that point. The string buffer is guaranteed to be null-terminated and will not contain the end-of-line characters.

There are some limitations when using **ReadLine**. The method should only be used to read text, never binary data. In particular, the method will discard nulls, linefeed and carriage return control characters. The Unicode version of this method will return a Unicode string, however it does not support reading raw Unicode data from the socket. Any data read from the socket is internally buffered as octets (eight-bit bytes) and converted to Unicode using the **MultiByteToWideChar** function.

This method will force the thread to block until an end-of-line character sequence is processed, the read operation times out or the remote host closes its end of the socket connection. If this

method is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, read the data and then restore the socket to asynchronous operation. If another socket operation is attempted while **ReadLine** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **Read** and **ReadLine** method calls can be intermixed, however be aware that **Read** will consume any data that has already been buffered by the **ReadLine** method and this may have unexpected results.

Unlike the **Read** method, it is possible for data to be returned in the string buffer even if the return value is zero. Applications should check the length of the string to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the return value.

Example

```
CString strBuffer;
BOOL bResult;

do
{
    bResult = pSocket->ReadLine(strBuffer);

    if (strBuffer.GetLength() > 0)
    {
        // Process the line of data returned in the string
        // buffer; the string is always null-terminated
    }
} while (bResult);

DWORD dwError = pSocket->GetLastError();

if (dwError == ST_ERROR_CONNECTION_CLOSED)
{
    // The remote host has closed its side of the connection and
    // there is no more data available to be read
}
else if (dwError != 0)
{
    // An error has occurred while reading a line of data
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IsReadable](#), [Peek](#), [Read](#), [ReadStream](#), [Write](#), [WriteLine](#), [WriteStream](#)

CSocketWrench::ReadStream Method

```
BOOL ReadStream(  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwOptions,  
    LPBYTE lpMarker,  
    DWORD cbMarker  
);
```

The **ReadStream** method reads the socket data stream and stores the contents in the specified buffer.

Parameters

lpvBuffer

Pointer to the buffer that will contain or reference the data when the method returns. The actual argument depends on the value of the *dwOptions* parameter which specifies how the data stream will be stored.

lpdwLength

A pointer to an unsigned integer value which specifies the maximum length of the buffer and contains the number of bytes read when the method returns. This argument should always point to an initialized value. If the *lpvBuffer* argument specifies a memory buffer, then this argument cannot point to an initialized value of zero; if any other type of stream buffer is used and the initialized value is zero, that indicates that all available data from the socket should be returned until the end-of-stream marker is encountered or the remote host disconnects.

dwOptions

An unsigned integer value which specifies both the stream buffer type and any options to be used when reading the data stream. One of the following stream types may be specified:

Constant	Description
INET_STREAM_DEFAULT	The default stream buffer type is determined by the value passed as the <i>lpvBuffer</i> parameter. If the argument specifies a pointer to a global memory handle initialized to NULL, then the method will return a handle which references the data; otherwise, the method will consider the parameter a pointer to a block of pre-allocated memory which will contain the stream data when the method returns. In most cases, it is recommended that an application explicitly specify the stream buffer type rather than using the default value.
INET_STREAM_MEMORY	The <i>lpvBuffer</i> argument specifies a pointer to a pre-allocated block of memory which will contain the data read from the socket when the method returns. If this stream buffer type is used, the <i>lpdwLength</i> argument must point to an unsigned integer which has been initialized with the maximum length of the buffer.

INET_STREAM_HGLOBAL	The <i>lpvBuffer</i> argument specifies a pointer to a global memory handle. When the method returns, the handle will reference a block of memory that contains the stream data. The application should take care to make sure that the handle passed to the method does not currently reference a valid block of memory; it is recommended that the handle be initialized to NULL prior to calling this method.
INET_STREAM_HANDLE	The <i>lpvBuffer</i> argument specifies a Windows handle to an open file, console or pipe. This should be the same handle value returned by the CreateFile function in the Windows API. The data read from the socket will be written to this handle using the WriteFile function.
INET_STREAM_SOCKET	The <i>lpvBuffer</i> argument specifies a socket handle. The data read from the socket specified by the <i>hSocket</i> argument will be written to this socket. The socket handle passed to this method must have been created by this library; if it is a socket created by an third-party library or directly by the Windows Sockets API, you should either create another instance of the class and attach the socket using the AttachHandle method or use the INET_STREAM_HANDLE stream buffer type instead.

In addition to the stream buffer types listed above, the *dwOptions* parameter may also have one or more of the following bit flags set. Programs should use a bitwise operator to combine values.

Constant	Description
INET_STREAM_CONVERT	The data stream is considered to be textual and will be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data returned in the buffer to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.
INET_STREAM_UNICODE	The data stream should be converted to Unicode. This option should only be used with text data, and will result in the stream data being returned as 16-bit wide characters rather than 8-bit bytes. The amount of data returned will be twice the amount

read from the source data stream; if the application is using a pre-allocated memory buffer, this must be considered before calling this method.
--

lpMarker

A pointer to an array of bytes which marks the end of the data stream. When this byte sequence is encountered by the method, it will stop reading and return to the caller. The buffer will contain all of the data read from the socket up to and including the end-of-stream marker. If this argument is NULL, then the method will continue to read from the socket until the maximum buffer size is reached, the remote host closes its socket or an error is encountered.

cbMarker

An unsigned integer value which specifies the length of the end-of-stream marker in bytes. If the *lpMarker* parameter is NULL, then this value must be zero.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ReadStream** method enables an application to read an arbitrarily large stream of data and store it in memory, write it to a file or even another socket. Unlike the **Read** method, which will return immediately when any amount of data has been read, **ReadStream** will only return when the buffer is full as specified by the *lpdwLength* parameter, the logical end-of-stream marker has been read, the socket closed by the remote host or when an error occurs.

This method will force the thread to block until the operation completes. If this method is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to asynchronous operation when it has finished. If another socket operation is attempted while **ReadStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

It is possible for data to be returned in the buffer even if the method returns a value of zero. Applications should also check the value of the *lpdwLength* argument to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the method return value.

Because **ReadStream** can potentially cause the application to block for long periods of time as the data stream is being read, the method will periodically generate INET_EVENT_PROGRESS events. An application can register an event handler using the **RegisterEvent** method, and can obtain information about the current operation by calling the **GetStreamInfo** method.

Example

```
HGLOBAL hglbBuffer = NULL; // Return data in a global memory buffer
```

```
DWORD cbBuffer = 102400; // Read up to 100K bytes
BOOL bResult;

bResult = pSocket->ReadStream(&hgblBuffer, &cbBuffer,
                             INET_STREAM_HGLOBAL | INET_STREAM_CONVERT);

if (bResult && cbBuffer > 0)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // Use data in the stream buffer

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

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[GetStreamInfo](#), [Read](#), [ReadLine](#), [StoreStream](#), [Write](#), [WriteLine](#), [WriteStream](#)

CSocketWrench::RegisterEvent Method

```
INT RegisterEvent(  
    UINT nEventId,  
    INETEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Constant	Description
INET_EVENT_CONNECT	The connection to the remote host has completed.
INET_EVENT_DISCONNECT	The remote host has closed the connection to the client. The client should read any remaining data and disconnect.
INET_EVENT_READ	A network event which indicates data is available to read. No additional messages will be posted until the process has read at least some of the data from the socket. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_WRITE	A network event which indicates the application can send data to the remote host. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The application may attempt to retry the operation, or may disconnect from the remote host and report an error to the user.
INET_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **InetEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is

INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **InetEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[DisableEvents](#), [EnableEvents](#), [FreezeEvents](#), [InetEventProc](#)

CSocketWrench::Reject Method

```
BOOL Reject();
```

The **Reject** method is used to reject a client connection request.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Reject** method rejects a pending client connection and the remote host will see this as the connection being aborted. If there are no pending client connections at the time, this method will immediately return with an error indicating that the operation would cause the thread to block.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[Accept](#), [Listen](#)

CSocketWrench::SetHostFile Method

```
INT SetHostFile(  
    LPCTSTR lpszFileName  
);
```

The **SetHostFile** method specifies the name of an alternate file to use when resolving hostnames and IP addresses. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **GetHostAddress** and **GetHostNames** method. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

```
ipaddress hostname [hostalias ...]
```

For example, one typical entry maps the name "localhost" to the local loopback IP address. This would be entered as:

```
127.0.0.1 localhost
```

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

Parameters

lpszFileName

Pointer to a string that specifies the name of the file. If the parameter is NULL, then the current host file is cleared from the cache and only the default host file will be used to resolve hostnames and addresses.

Return Value

If the method succeeds, the return value is the number of entries in the host file. A return value of INET_ERROR indicates failure. To get extended error information, call **GetLastError**.

Remarks

This method loads the file into memory allocated for the current thread. If the contents of the file have changed after the method has been called, those changes will not be reflected when resolving hostnames or addresses. To reload the host file from disk, call this method again with the same file name. To remove the alternate host file from memory, specify a NULL pointer as the parameter.

If a host file has been specified, it is processed before the default host file when resolving a hostname into an IP address, or an IP address into a hostname. If the host name or address is not found, or no host file has been specified, a nameserver lookup is performed.

To determine if an alternate host file has been specified, use the **GetHostFile** method. A return value of zero indicates that no alternate host file has been cached for the current thread.

A system may have a default host file, which is used to resolve hostnames before performing a nameserver lookup. To determine the name of this file, use the **GetDefaultHostFile** method. It is not necessary to specify this default host file, since it is always used to resolve host names and addresses.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cswsock10.h

Import Library: csWSKV10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetDefaultHostFile](#), [GetHostAddress](#), [GetHostFile](#), [GetHostName](#)

CSocketWrench::SetLastError Method

```
VOID SetLastError(  
    DWORD dwErrorCode  
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SOCKET or INET_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[GetErrorString](#), [GetLastError](#), [ShowError](#)

CSocketWrench::SetOption Method

```
INT SetOption(  
    DWORD dwOption,  
    BOOL bEnabled  
);
```

The **SetOption** method is used to enable or disable a specific socket option.

Parameters

dwOption

An unsigned integer used to specify one of the socket options. These options cannot be combined. The following values are recognized:

Constant	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_KEEPAIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option is commonly used by server applications.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.

bEnabled

A boolean flag. If the flag is set to a non-zero value, the option is enabled. Otherwise the socket option is disabled.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

It is not recommend that you disable the Nagle algorithm by specifying the INET_OPTION_NODELAY flag unless it is absolutely required. Doing so can have a significant, negative impact on the performance of the application and network.

If if the INET_OPTION_KEEPAIVE option is enabled, keep-alive packets will start being generated five seconds after the socket has become idle with no data being sent or received. Enabling this option can be used by applications to detect when a physical network connection has been lost. However, it is recommended that most applications query the remote host directly to determine if the connection is still active. This is typically accomplished by sending specific commands to the server to query its status, or checking the elapsed time since the last response from the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cswsock10.h

Import Library: cswskv10.lib

See Also

[Connect](#), [GetOption](#)

CSocketWrench::SetTimeout Method

```
INT SetTimeout(  
    UINT nTimeout  
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the remote host. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[Connect](#), [GetTimeout](#), [IsReadable](#), [IsWritable](#), [Read](#), [Write](#)

CSocketWrench::ShowError Method

```
INT ShowError(  
    LPCTSTR lpszAppTitle,  
    UINT uType,  
    DWORD dwErrorCode  
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A pointer to a string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetErrorString](#), [GetLastError](#), [SetLastError](#)

CSocketWrench::Shutdown Method

```
INT Shutdown(  
    DWORD dwOption  
);
```

The **Shutdown** method is used to disable reception or transmission of data, or both.

Parameters

dwOption

An unsigned integer used to specify one of the shutdown options. These options cannot be combined. The following values are recognized:

Value	Constant	Description
0	INET_SHUTDOWN_READ	Disable reception of data.
1	INET_SHUTDOWN_WRITE	Disable transmission of data.
2	INET_SHUTDOWN_BOTH	Disable both reception and transmission of data.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method is rarely needed. It is provided as an interface to the Windows Sockets **shutdown** method.

In some asynchronous applications, it may be desirable for a client to inform the server that no further communication is wanted, while allowing the client to read any residual data that may reside in internal buffers on the client side. **Shutdown** accomplishes this because the socket handle is still valid after it has been called, although some or all communication with the remote host has ceased.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[Abort](#), [Connect](#), [Disconnect](#)

CSocketWrench::StoreStream Method

```
BOOL StoreStream(  
    LPCTSTR lpszFileName,  
    DWORD dwLength,  
    LPDWORD lpdwCopied  
    DWORD dwOffset,  
    DWORD dwOptions  
);
```

The **StoreStream** method reads the socket data stream and stores the contents in the specified file.

Parameters

lpszFileName

Pointer to a string which specifies the name of the file to create or overwrite.

dwLength

An unsigned integer which specifies the maximum number of bytes to read from the socket and write to the file. If this value is zero, then the method will continue to read data from the socket until the remote host disconnects or an error occurs.

lpdwCopied

A pointer to an unsigned integer value which will contain the number of bytes written to the file when the method returns.

dwOffset

An unsigned integer which specifies the byte offset into the file where the method will start storing data read from the socket. Note that all data after this offset will be truncated. A value of zero specifies that the file should be completely overwritten if it already exists.

dwOptions

An unsigned integer value which specifies one or more options. Programs can use a bitwise operator to combine any of the following values:

Constant	Description
INET_STREAM_CONVERT	The data stream is considered to be textual and will be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data written to the file to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.
INET_STREAM_UNICODE	The data stream should be converted to Unicode. This option should only be used with text data, and will result in the stream data being written as 16-bit

wide characters rather than 8-bit bytes. The amount of data returned will be twice the amount read from the source data stream. If the *dwOffset* parameter has a value of zero, the Unicode byte order mark (BOM) will be written to the beginning of the file.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **StoreStream** method enables an application to read an arbitrarily large stream of data and store it in a file. This method is essentially a simplified version of the **ReadStream** method, designed specifically to be used with files rather than memory buffers or handles.

This method will force the thread to block until the operation completes. If this method is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to asynchronous operation when it has finished. If another socket operation is attempted while **StoreStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

Because **StoreStream** can potentially cause the application to block for long periods of time as the data stream is being read, the method will periodically generate `INET_EVENT_PROGRESS` events. An application can register an event handler using the **RegisterEvent** method, and can obtain information about the current operation by calling the **GetStreamInfo** method.

Example

```
DWORD dwCopied;
BOOL bResult;

bResult = pSocket->StoreStream(lpszFileName, 0, &dwCopied, 0,
                               INET_STREAM_CONVERT);

if (bResult && dwCopied > 0)
{
    // The data has been written to the file
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetStreamInfo](#), [Read](#), [ReadLine](#), [ReadStream](#), [Write](#), [WriteLine](#), [WriteStream](#)

CSocketWrench::Write Method

```
INT Write(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Write(  
    LPBYTE lpBuffer,  
    INT cbBuffer,  
    LPINTERNET_ADDRESS lpAddress,  
    UINT nRemotePort  
);
```

The **Write** method sends the specified number of bytes to the remote host.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the remote host.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero.

lpAddress

Pointer to an [INTERNET_ADDRESS](#) structure that specifies the address of the remote host that is to receive the data being written. For TCP stream sockets, this parameter must always be NULL or specify the same address that was used to establish the connection. For UDP datagram sockets, this may specify any valid IP address.

nRemotePort

The port number of the remote host that is to receive the data being written. For TCP stream sockets, this value must always be zero, or specify the same port number that was used to establish the connection. For UDP datagram sockets, this may specify any valid port number.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking sockets, the client must wait for the next asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableEvents](#), [IsBlocking](#), [IsReadable](#), [IsWritable](#), [Read](#), [ReadLine](#), [RegisterEvent](#), [WriteLine](#)

CSocketWrench::WriteLine Method

```
BOOL WriteLine(  
    LPCTSTR lpszBuffer,  
    LPINT lpnLength  
);
```

The **WriteLine** function sends a line of text to the remote host, terminated by a carriage-return and linefeed.

Parameters

lpszBuffer

The pointer to a string buffer which contains the data that will be sent to the remote host. All characters up to, but not including, the terminating null character will be written to the socket. The data will always be terminated with a carriage-return and linefeed control character sequence. If this parameter points to an empty string or NULL pointer, then a only a carriage-return and linefeed are written to the socket.

lpnLength

A pointer to an integer value which will contain the number of characters written to the socket, including the carriage-return and linefeed sequence. If this information is not required, a NULL pointer may be specified.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **WriteLine** method writes a line of text to the remote host and terminates the line with a carriage-return and linefeed control character sequence. Unlike the **Write** method which writes arbitrary bytes of data to the socket, this method is specifically designed to write a single line of text data from a null-terminated string.

If the *lpszBuffer* string is terminated with a linefeed (LF) or carriage return (CR) character, it will be automatically converted to a standard CRLF end-of-line sequence. Because the string will be sent with a terminating CRLF sequence, the value returned in the *lpnLength* parameter will typically be larger than the original string length (reflecting the additional CR and LF characters), unless the string was already terminated with CRLF.

There are some limitations when using **WriteLine**. This method should only be used to send text, never binary data. In particular, it will discard nulls and append linefeed and carriage return control characters to the data stream. The Unicode version of this method will accept a Unicode string, however it does not support writing raw Unicode data to the socket. Unicode strings will be automatically converted to UTF-8 encoding using the **WideCharToMultiByte** function and then written to the socket as a stream of bytes.

This method will force the thread to block until the complete line of text has been written, the write operation times out or the remote host aborts the connection. If this method is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, send the data and then restore the socket to asynchronous operation. If another socket operation is attempted while **WriteLine** is blocked sending data to the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker

threads to manage each connection.

The **Write** and **WriteLine** methods can be safely intermixed.

Unlike the **Write** function, it is possible for data to have been written to the socket if the return value is zero. For example, if a timeout occurs while the method is waiting to send more data to the remote host, it will return zero; however, some data may have already been written prior to the error condition. If this is the case, the *lpnLength* argument will specify the number of characters actually written up to that point.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IsWritable](#), [Read](#), [ReadLine](#), [ReadStream](#), [Write](#), [WriteStream](#)

CSocketWrench::WriteStream Method

```
BOOL WriteStream(  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwOptions  
);
```

The **WriteStream** method writes data from the stream buffer to the specified socket.

Parameters

lpvBuffer

Pointer to the buffer that contains or references the data to be written to the socket. The actual argument depends on the value of the *dwOptions* parameter which specifies how the data stream will be accessed.

lpdwLength

A pointer to an unsigned integer value which specifies the size of the buffer and contains the number of bytes written when the method returns. This argument should always point to an initialized value. If the *lpvBuffer* argument specifies a memory buffer or global memory handle, then this argument cannot point to an initialized value of zero.

dwOptions

An unsigned integer value which specifies the stream buffer type to be used when writing the data stream to the socket. One of the following stream types may be specified:

Constant	Description
INET_STREAM_DEFAULT	The default stream buffer type is determined by the value passed as the <i>lpvBuffer</i> parameter. If the argument specifies a global memory handle, then the method will write the data referenced by that handle; otherwise, the method will consider the parameter a pointer to a block of memory which contains data to be written. In most cases, it is recommended that an application explicitly specify the stream buffer type rather than using the default value.
INET_STREAM_MEMORY	The <i>lpvBuffer</i> argument specifies a pointer to a block of memory which contains the data to be written to the socket. If this stream buffer type is used, the <i>lpdwLength</i> argument must point to an unsigned integer which has been initialized with the size of the buffer.
INET_STREAM_HGLOBAL	The <i>lpvBuffer</i> argument specifies a global memory handle that references the data to be written to the socket. The handle must have been created by a call to the GlobalAlloc or GlobalReAlloc function. If this stream buffer type is used, the <i>lpdwLength</i> argument must point to an unsigned integer which has been initialized with the size of the buffer.

INET_STREAM_HANDLE	The <i>lpvBuffer</i> argument specifies a Windows handle to an open file, console or pipe. This should be the same handle value returned by the CreateFile function in the Windows API. The data read using the ReadFile function with this handle will be written to the socket.
INET_STREAM_SOCKET	The <i>lpvBuffer</i> argument specifies a socket handle. The data read from the socket specified by this handle will be written to the socket specified by the <i>hSocket</i> parameter. The socket handle passed to this method must have been created by this library; if it is a socket created by an third-party library or directly by the Windows Sockets API, you should either create another instance of the class and attach the socket using the AttachHandle method or use the INET_STREAM_HANDLE stream buffer type instead.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **WriteStream** method enables an application to write an arbitrarily large stream of data from memory or a file to the specified socket. Unlike the **Write** method, which may not write all of the data in a single method call, **WriteStream** will only return when all of the data has been written or an error occurs.

This method will force the thread to block until the operation completes. If this method is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, write the data stream and then restore the socket to asynchronous operation when it has finished. If another socket operation is attempted while **WriteStream** is blocked sending data to the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

It is possible for some data to have been written even if the method returns a value of zero. Applications should also check the value of the *lpdwLength* argument to determine if any data was sent. For example, if a timeout occurs while the method is waiting to write more data, it will return zero; however, some data may have already been written to the socket prior to the error condition.

Because **WriteStream** can potentially cause the application to block for long periods of time as the data stream is being written, the method will periodically generate INET_EVENT_PROGRESS events. An application can register an event handler using the **RegisterEvent** method, and can obtain information about the current operation by calling the **GetStreamInfo** method.

Example

```
CFile *pFile = new CFile();
DWORD dwLength = 0;
```

```
if (!pFile->Open(strFileName, CFile::modeRead | CFile::shareDenyWrite))
    return;

dwLength = pFile->GetLength();

if (dwLength > 0)
{
    BOOL bResult = pSocket->WriteStream(
        pFile->m_hFile,
        &dwLength,
        INET_STREAM_HANDLE);
}

delete pFile;
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[GetStreamInfo](#), [Read](#), [ReadLine](#), [ReadStream](#), [StoreStream](#), [Write](#), [WriteLine](#)

SocketWrench Data Structures

- INETSTREAMINFO
- INTERNET_ADDRESS
- SECURITYCREDENTIALS
- SECURITYINFO

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

INETSTREAMINFO Structure

This structure contains information about the data stream being currently read or written.

```
typedef struct _INETSTREAMINFO
{
    DWORD    dwStreamThread;
    DWORD    dwStreamSize;
    DWORD    dwStreamCopied;
    DWORD    dwStreamMode;
    DWORD    dwStreamError;
    DWORD    dwBytesPerSecond;
    DWORD    dwTimeElapsed;
    DWORD    dwTimeEstimated;
} INETSTREAMINFO, *LPINETSTREAMINFO;
```

Members

dwStreamThread

Specifies the numeric ID for the thread that created the socket.

dwStreamSize

The maximum number of bytes that will be read or written. This is the same value as the buffer length specified by the caller, and may be zero which indicates that no maximum size was specified. Note that if this value is zero, the application will be unable to calculate a completion percentage or estimate the amount of time for the operation to complete.

dwStreamCopied

The total number of bytes that have been copied to or from the stream buffer.

dwStreamMode

A numeric value which specifies the stream operation that is current being performed. It may be one of the following values:

Constant	Description
INET_STREAM_READ	Data is being read from the socket and stored in the specified stream buffer.
INET_STREAM_WRITE	Data is being written from the specified stream buffer to the socket.

dwStreamError

The last error that occurred when reading or writing the data stream. If no error has occurred, this value will be zero.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the operation is completed. This is based on the average number of bytes transferred per second and requires that a maximum stream buffer size be specified by the caller.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

See Also

[ReadStream](#), [StoreStream](#), [WriteStream](#)

INTERNET_ADDRESS Structure

This structure represents a numeric IPv4 or IPv6 address in network byte order.

```
typedef struct _INTERNET_ADDRESS
{
    INT    ipFamily;
    BYTE   ipNumber[16];
} INTERNET_ADDRESS, *LPINTERNET_ADDRESS;
```

Members

ipFamily

An integer which identifies the type of IP address. It will be one of the following values:

Constant	Description
INET_ADDRESS_UNKNOWN	The address has not been specified or the bytes in the <i>ipNumber</i> array does not represent a valid address. Functions which populate this structure will use this value to indicate that the address cannot be determined.
INET_ADDRESS_IPV4	Specifies that the address is in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address is in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

ipNumber

A byte array which contains the numeric form of the IP address. This array is large enough to store both IPv4 (32 bit) and IPv6 (128 bit) addresses. The values are stored in network byte order.

Remarks

The **INTERNET_ADDRESS** structure is used by some functions to represent an Internet address in a binary format that is compatible with both IPv4 and IPv6 addresses. Applications that use this structure should consider it to be opaque, and should not modify the contents of the structure directly.

For compatibility with legacy applications that expect an IP address to be 32 bits and stored in an unsigned integer, you can copy the first four bytes of the *ipNumber* array using the **CopyMemory** function or equivalent. Note that if this is done, your application should always check the *ipFamily* member first to make sure that it is actually an IPv4 address.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

SECURITYCREDENTIALS Structure

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on

	what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that

will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A pointer to a null terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A pointer to a null terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A pointer to a null terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A pointer to a null terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
------------	-------------

CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
----	---

MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
----	--

ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.
------	--

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

A pointer to a null terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used for SSH connections

and should always be NULL when establishing a secure connection using SSL or TLS.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* member should be set to CREDENTIAL_STORE_FILENAME and the *lpzCertStore* member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS#12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the *lpzPassword* member or the library will be unable to access the certificate.

Note that the *lpzUserName* and *lpzPassword* members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

The TLS 1.1 and TLS 1.2 protocols are only supported on Windows 7, Windows Server 2008 R2 and later versions of the platform. If these options are specified and the application is running on Windows XP or Windows Vista, the protocol version will be downgraded to TLS 1.0 for backwards compatibility with those platforms.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Unicode: Implemented as Unicode and ANSI versions.

SECURITYINFO Structure

This structure contains information about a secure connection that has been established.

```
typedef struct _SECURITYINFO
{
    DWORD        dwSize;
    DWORD        dwProtocol;
    DWORD        dwCipher;
    DWORD        dwCipherStrength;
    DWORD        dwHash;
    DWORD        dwHashStrength;
    DWORD        dwKeyExchange;
    DWORD        dwCertStatus;
    SYSTEMTIME   stCertIssued;
    SYSTEMTIME   stCertExpires;
    LPCTSTR      lpszCertIssuer;
    LPCTSTR      lpszCertSubject;
    LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to **sizeof(SECURITYINFO)** prior to passing the address of this structure to the function. Note that if this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the protocol that was selected to establish the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The

	<p>correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.</p>
SECURITY_PROTOCOL_TLS10	<p>The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS11	<p>The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS12	<p>The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.</p>

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_CIPHER_RC2	The RC2 block cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
SECURITY_CIPHER_DES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher, using 56-bit

	keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher (also known as the Rijndael cipher) is a fixed block size cipher which use a key size of 128, 192 or 256 bits. This cipher is supported on Windows XP SP3 and later versions of the operating system.
SECURITY_CIPHER_SKIPJACK	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
SECURITY_CIPHER_BLOWFISH	The Blowfish block cipher was selected. This is a variable key length cipher up to 448 bits, using a 64-bit block size.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected. One of the following values will be returned:

Constant	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected. One of the following values will be returned:

--	--

Constant	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography. This key exchange algorithm is only supported on Windows XP SP3 and later versions of the operating system.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server. This member only has meaning for connections using the SSL or TLS protocols. One of the following values will be returned:

Constant	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertIssuer

A pointer to a string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertSubject

A pointer to a string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszFingerprint

A pointer to a string which contains a sequence of hexadecimal values that uniquely identify the server. This member is only used when a connection has been established using the Secure Shell (SSH) protocol.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Unicode: Implemented as Unicode and ANSI versions.

SocketWrench Class Error Codes

Value	Constant	Description
0x80042711	ST_ERROR_NOT_HANDLE_OWNER	Handle not owned by the current thread
0x80042712	ST_ERROR_FILE_NOT_FOUND	The specified file or directory does not exist
0x80042713	ST_ERROR_FILE_NOT_CREATED	The specified file could not be created
0x80042714	ST_ERROR_OPERATION_CANCELED	The blocking operation has been canceled
0x80042715	ST_ERROR_INVALID_FILE_TYPE	The specified file is a block or character device, not a regular file
0x80042716	ST_ERROR_INVALID_DEVICE	The specified device or address does not exist
0x80042717	ST_ERROR_TOO_MANY_PARAMETERS	The maximum number of method parameters has been exceeded
0x80042718	ST_ERROR_INVALID_FILE_NAME	The specified file name contains invalid characters or is too long
0x80042719	ST_ERROR_INVALID_FILE_HANDLE	Invalid file handle passed to method
0x8004271A	ST_ERROR_FILE_READ_FAILED	Unable to read data from the specified file
0x8004271B	ST_ERROR_FILE_WRITE_FAILED	Unable to write data to the specified file
0x8004271C	ST_ERROR_OUT_OF_MEMORY	Out of memory
0x8004271D	ST_ERROR_ACCESS_DENIED	Access denied
0x8004271E	ST_ERROR_INVALID_PARAMETER	Invalid argument passed to method
0x8004271F	ST_ERROR_CLIPBOARD_UNAVAILABLE	The system clipboard is currently unavailable
0x80042720	ST_ERROR_CLIPBOARD_EMPTY	The system clipboard is empty or does not contain any text data
0x80042721	ST_ERROR_FILE_EMPTY	The specified file does not contain any data
0x80042722	ST_ERROR_FILE_EXISTS	The specified file already exists
0x80042723	ST_ERROR_END_OF_FILE	End of file
0x80042724	ST_ERROR_DEVICE_NOT_FOUND	The specified device could not be found
0x80042725	ST_ERROR_DIRECTORY_NOT_FOUND	The specified directory could not be found
0x80042726	ST_ERROR_INVALID_BUFFER	Invalid memory address passed to method
0x80042728	ST_ERROR_NO_HANDLES	No more handles available to this process
0x80042733	ST_ERROR_OPERATION_WOULD_BLOCK	The specified operation would block the current thread
0x80042734	ST_ERROR_OPERATION_IN_PROGRESS	A blocking operation is currently in progress
0x80042735	ST_ERROR_ALREADY_IN_PROGRESS	The specified operation is already in progress
0x80042736	ST_ERROR_INVALID_HANDLE	Invalid handle passed to method

0x80042737	ST_ERROR_INVALID_ADDRESS	Invalid network address specified
0x80042738	ST_ERROR_INVALID_SIZE	Datagram is too large to fit in specified buffer
0x80042739	ST_ERROR_INVALID_PROTOCOL	Invalid network protocol specified
0x8004273A	ST_ERROR_PROTOCOL_NOT_AVAILABLE	The specified network protocol is not available
0x8004273B	ST_ERROR_PROTOCOL_NOT_SUPPORTED	The specified protocol is not supported
0x8004273C	ST_ERROR_SOCKET_NOT_SUPPORTED	The specified socket type is not supported
0x8004273D	ST_ERROR_INVALID_OPTION	The specified option is invalid
0x8004273E	ST_ERROR_PROTOCOL_FAMILY	Specified protocol family is not supported
0x8004273F	ST_ERROR_PROTOCOL_ADDRESS	The specified address is invalid for this protocol family
0x80042740	ST_ERROR_ADDRESS_IN_USE	The specified address is in use by another process
0x80042741	ST_ERROR_ADDRESS_UNAVAILABLE	The specified address cannot be assigned
0x80042742	ST_ERROR_NETWORK_UNAVAILABLE	The networking subsystem is unavailable
0x80042743	ST_ERROR_NETWORK_UNREACHABLE	The specified network is unreachable
0x80042744	ST_ERROR_NETWORK_RESET	Network dropped connection on remote reset
0x80042745	ST_ERROR_CONNECTION_ABORTED	Connection was aborted due to timeout or other failure
0x80042746	ST_ERROR_CONNECTION_RESET	Connection was reset by remote network
0x80042747	ST_ERROR_OUT_OF_BUFFERS	No buffer space is available
0x80042748	ST_ERROR_ALREADY_CONNECTED	Connection already established with remote host
0x80042749	ST_ERROR_NOT_CONNECTED	No connection established with remote host
0x8004274A	ST_ERROR_CONNECTION_SHUTDOWN	Unable to send or receive data after connection shutdown
0x8004274C	ST_ERROR_OPERATION_TIMEOUT	The specified operation has timed out
0x8004274D	ST_ERROR_CONNECTION_REFUSED	The connection has been refused by the remote host
0x80042750	ST_ERROR_HOST_UNAVAILABLE	The specified host is unavailable
0x80042751	ST_ERROR_HOST_UNREACHABLE	Remote host is unreachable
0x80042753	ST_ERROR_TOO_MANY_PROCESSES	Too many processes are using the networking subsystem
0x8004276B	ST_ERROR_NETWORK_NOT_READY	Network subsystem is not ready for communication
0x8004276C	ST_ERROR_INVALID_VERSION	This version of the operating system is not supported

0x8004276D	ST_ERROR_NETWORK_NOT_INITIALIZED	The networking subsystem has not been initialized
0x80042775	ST_ERROR_REMOTE_SHUTDOWN	The remote host has initiated a graceful shutdown sequence
0x80042AF9	ST_ERROR_INVALID_HOSTNAME	The specified hostname is invalid or could not be resolved
0x80042AFA	ST_ERROR_HOSTNAME_NOT_FOUND	The specified hostname could not be found
0x80042AFB	ST_ERROR_HOSTNAME_REFUSED	Unable to resolve hostname, request refused
0x80042AFC	ST_ERROR_HOSTNAME_NOT_RESOLVED	Unable to resolve hostname, no address for specified host
0x80042EE1	ST_ERROR_INVALID_LICENSE	The license for this product is invalid
0x80042EE2	ST_ERROR_PRODUCT_NOT_LICENSED	This product is not licensed to perform this operation
0x80042EE3	ST_ERROR_NOT_IMPLEMENTED	This method has not been implemented on this platform
0x80042EE4	ST_ERROR_UNKNOWN_LOCALHOST	Unable to determine local host name
0x80042EE5	ST_ERROR_INVALID_HOSTADDRESS	Invalid host address specified
0x80042EE6	ST_ERROR_INVALID_SERVICE_PORT	Invalid service port number specified
0x80042EE7	ST_ERROR_INVALID_SERVICE_NAME	Invalid or unknown service name specified
0x80042EE8	ST_ERROR_INVALID_EVENTID	Invalid event identifier specified
0x80042EE9	ST_ERROR_OPERATION_NOT_BLOCKING	No blocking operation in progress on this socket
0x80042F45	ST_ERROR_SECURITY_NOT_INITIALIZED	Unable to initialize security interface for this process
0x80042F46	ST_ERROR_SECURITY_CONTEXT	Unable to establish security context for this session
0x80042F47	ST_ERROR_SECURITY_CREDENTIALS	Unable to open client certificate store or establish client credentials
0x80042F48	ST_ERROR_SECURITY_CERTIFICATE	Unable to validate the certificate chain for this session
0x80042F49	ST_ERROR_SECURITY_DECRYPTION	Unable to decrypt data stream
0x80042F4A	ST_ERROR_SECURITY_ENCRYPTION	Unable to encrypt data stream
0x80043031	ST_ERROR_MAXIMUM_CONNECTIONS	The maximum number of client connections exceeded
0x80043032	ST_ERROR_THREAD_CREATION_FAILED	Unable to create a new thread for the current process
0x80043033	ST_ERROR_INVALID_THREAD_HANDLE	The specified thread handle is no longer valid
0x80043034	ST_ERROR_THREAD_TERMINATED	The specified thread has been terminated
0x80043035	ST_ERROR_THREAD_DEADLOCK	The operation would result in the current

		thread becoming deadlocked
0x80043036	ST_ERROR_INVALID_CLIENT_MONIKER	The specified moniker is not associated with any client session
0x80043037	ST_ERROR_CLIENT_MONIKER_EXISTS	The specified moniker has been assigned to another client session
0x80043038	ST_ERROR_SERVER_INACTIVE	The specified server is not listening for client connections
0x80043039	ST_ERROR_SERVER_SUSPENDED	The specified server is suspended and not accepting client connections

Telnet Protocol Class Library

Establish an interactive terminal session with a server.

Reference

- [Class Methods](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

Class Name	CTelnetClient
File Name	CSTNTV10.DLL
Version	10.0.1468.2518
LibID	A081335C-B907-4179-949F-FCE154F7C456
Import Library	CSTNTV10.LIB
Dependencies	None
Standards	RFC 854

Overview

The Telnet protocol is used to establish a connection with a server which provides a virtual terminal session for a user. Its functionality is similar to how character based consoles and serial terminals work, enabling a user to login to the server, execute commands and interact with applications running on the server. The class provides an interface for establishing the connection, negotiating certain options (such as whether characters will be echoed back to the client) and handling the standard I/O functions needed by the program.

The class also provides methods that enable a program to easily scan the data stream for specific sequences of characters, making it very simple to write light-weight client interfaces to applications running on the server. The CTelnetClient class can be combined with the CTerminalEmulator class to provide complete terminal emulation services for a standard ANSI or DEC-VT220 terminal.

This class supports secure connections using the standard SSL and TLS protocols. To establish a secure connection to the server using the Secure Shell (SSH) protocol, use the SocketTools CSshClient class.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-

bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Telnet Protocol Class Methods

Class	Description
CTelnetClient	Constructor which initializes the current instance of the class
~CTelnetClient	Destructor which releases resources allocated by the class
Method	Description
Abort	Abort the current session and close the connection with the server
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified client handle to another thread
Break	Send a break signal to the server
Cancel	Cancel the current blocking operation
Connect	Connect to the specified server
CreateSecurityCredentials	Create a new security credentials structure
DeleteSecurityCredentials	Delete a previously created security credentials structure
DetachHandle	Detach the handle for the current instance of this class
DisableEvents	Disable asynchronous event notification
DisableTrace	Disable logging of socket function calls to the trace log
Disconnect	Disconnect from the current server
EnableEvents	Enable asynchronous event notification
EnableTrace	Enable logging of socket function calls to a file
FreezeEvents	Suspend asynchronous event processing
GetErrorString	Return a description for the specified error code
GetHandle	Return the client handle used by this instance of the class
GetLastError	Return the last error code
GetMode	Return the current client mode
GetSecurityInformation	Return security information about the current client connection
GetStatus	Return the current client status
GetTerminalType	Return the current terminal type
GetTimeout	Return the number of seconds until an operation times out
IsBlocking	Determine if the client is blocked, waiting for information
IsConnected	Determine if the client is connected to the server
IsInitialized	Determine if the class has been successfully initialized
IsReadable	Determine if data can be read from the server
IsThere	Determine if the server is available
IsWritable	Determine if data can be written to the server

Login	Login to the server using the specified username and password
Read	Read data returned by the server
ReadLine	Read a line of text from the server and return it in a string buffer
RegisterEvent	Register an event callback function
Search	Search for a specific character sequence in the data stream
SetLastError	Set the last error code
SetMode	Set the current client mode
SetTerminalType	Set the current terminal type
SetTimeout	Set the number of seconds until an operation times out
ShowError	Display a message box with a description of the specified error
TelnetEventProc	Callback function that processes events generated by the client
Write	Write data to the server
WriteLine	Write a line of text to the server

CTelnetClient::CTelnetClient Method

`CTelnetClient();`

The **CTelnetClient** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **TelnetInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstntv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[~CTelnetClient](#), [IsInitialized](#)

CTelnetClient::~CTelnetClient

`~CTelnetClient();`

The **CTelnetClient** destructor releases resources allocated by the current instance of the **CTelnetClient** object. It also uninitialized the library if there are no other concurrent uses of the class.

Remarks

When a **CTelnetClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstntv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CTelnetClient](#)

CTelnetClient::Abort Method

```
INT Abort();
```

The **Abort** method aborts the current session and terminates the connection.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Abort** method is called, the abort sequence is sent to the server and the connection to the server is terminated. Once this method returns, the client handle is no longer valid. If a program is currently executing on the server at the time this method is called, that program may be terminated as a result of the session being aborted. Applications should normally call **Disconnect** to gracefully disconnect from the server and should only use this method when the connection must be aborted immediately.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[Break](#), [Cancel](#), [IsBlocking](#)

CTelnetClient::AttachHandle Method

```
VOID AttachHandle(  
    HCLIENT hClient  
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle already has been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the *hClient* parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[AttachThread](#), [DetachHandle](#), [GetHandle](#)

CTelnetClient::AttachThread Method

```
DWORD AttachThread(  
    DWORD dwThreadId  
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstntv10.lib

See Also

[AttachHandle](#), [Cancel](#), [Connect](#), [DetachHandle](#), [Disconnect](#), [GetHandle](#)

CTelnetClient::Break Method

INT Break();

The **Break** method sends a signal to the server which may terminate an application that is currently running. The actual response to the break signal depends on the application.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Abort](#), [Cancel](#), [Read](#), [Write](#)

CTelnetClient::Cancel Method

```
INT Cancel();
```

The **Cancel** method cancels any outstanding blocking operation in the client, causing the blocking method to fail. The application may then retry the operation or terminate the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[Abort](#), [Break](#), [IsBlocking](#)

CTelnetClient::Connect Method

```
BOOL Connect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **Connect** method establishes a connection with the specified server.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used. For standard connections, the default port number is 23. For secure connections, the default port number is 992.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TELNET_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
TELNET_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
TELNET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
TELNET_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server.

	Note that the server must support secure connections using either the SSL or TLS protocol.
TELNET_OPTION_SECURE_EXPLICIT	This option specifies the client should attempt to establish a secure connection with the server using the START_TLS option. The client initiates a standard connection with the server, then requests a secure connection during the option negotiation process.
TELNET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
TELNET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
TELNET_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur. If this argument is NULL, then the client session will be blocking and no network events will be sent to the client.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the *hEventWnd* argument is NULL, this argument should be specified as WM_NULL.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The use of Windows event notification messages has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the

application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

To prevent this method from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **Connect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

If you specify an event notification window, then the client session will be asynchronous. When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
TELNET_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
TELNET_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
TELNET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
TELNET_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
TELNET_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
TELNET_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To cancel asynchronous notification and return the client to a blocking mode, use the **DisableEvents** methods.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the class instance is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call methods using

that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the TELNET_OPTION_FREETHREAD option enables any thread to call any method in that instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same instance.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateSecurityCredentials](#), [Disconnect](#), [GetSecurityInformation](#)

CTelnetClient::CreateSecurityCredentials Method

```
BOOL CreateSecurityCredentials(  
    DWORD dwProtocol,  
    DWORD dwOptions,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName  
);  
  
BOOL CreateSecurityCredentials(  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName  
);  
  
BOOL CreateSecurityCredentials(  
    LPCTSTR lpszCertName  
);
```

The **CreateSecurityCredentials** method establishes the security credentials for the client session.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols.

	This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpzUserName

A pointer to a string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A pointer to a string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A pointer to a string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Example

```
pClient->CreateSecurityCredentials(  
    SECURITY_PROTOCOL_DEFAULT,  
    0,  
    NULL,  
    NULL,  
    lpszCertStore,  
    lpszCertName);
```

```
bConnected = pClient->Connect(lpszHostName,  
                              TELNET_PORT_SECURE,  
                              TELNET_TIMEOUT,  
                              TELNET_OPTION_SECURE);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [DeleteSecurityCredentials](#), [GetSecurityInformation](#), [SECURITYCREDENTIALS](#)

CTelnetClient::DeleteSecurityCredentials Method

```
VOID DeleteSecurityCredentials();
```

The **DeleteSecurityCredentials** method releases the security credentials for the current session.

Parameters

None.

Return Value

None.

Remarks

This method can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated. The security credentials are released when the class destructor is called, so it is normally not required that the application explicitly call this method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CreateSecurityCredentials](#)

CTelnetClient::DetachHandle Method

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstntv10.lib`

See Also

[AttachHandle](#), [GetHandle](#)

CTelnetClient::DisableEvents Method

```
INT DisableEvents();
```

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the client has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[EnableEvents](#), [RegisterEvent](#)

CTelnetClient::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[EnableTrace](#)

CTelnetClient::Disconnect Method

VOID Disconnect();

The **Disconnect** method terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[Connect](#), [IsConnected](#)

CTelnetClient::EnableEvents Method

```
INT EnableEvents(  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Constant	Description
TELNET_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
TELNET_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
TELNET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
TELNET_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking

	operation. This event is only generated if the client is in asynchronous mode.
TELNET_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
TELNET_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[DisableEvents](#), [RegisterEvent](#)

CTelnetClient::EnableTrace Method

```
BOOL EnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **EnableTrace** method enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace method logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableTrace](#)

CTelnetClient::FreezeEvents Method

```
INT FreezeEvents(  
    BOOL bFreeze  
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the client.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableEvents](#), [EnableEvents](#), [RegisterEvent](#)

CTelnetClient::GetErrorString Method

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);  
  
INT GetErrorString(  
    DWORD dwErrorCode,  
    CString& strDescription  
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLastError](#), [SetLastError](#), [ShowError](#)

CTelnetClient::GetHandle Method

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstntv10.lib`

See Also

[AttachHandle](#), [DetachHandle](#), [IsInitialized](#)

CTelnetClient::GetLastError Method

```
DWORD GetLastError();  
  
DWORD GetLastError(  
    CString& strDescription  
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError(0)** when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or TELNET_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cstntv10.lib

See Also

[GetErrorString](#), [SetLastError](#), [ShowError](#)

CTelnetClient::GetMode Method

```
UINT GetMode();
```

The **GetMode** method returns the current client mode.

Parameters

None.

Return Values

If the method succeeds, the return value is the current client mode. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The client mode is a combination of one or more flags which determines how the client handles local character echo and character processing. The following values are recognized:

Value	Description
TELNET_MODE_LOCALECHO	The local client is responsible for echoing data entered by the user. By default, this mode is not set which means that the server is responsible for echoing back each character written to it.
TELNET_MODE_BINARY	Data exchanged between the client and server should not be converted or line buffered. If this option is not specified, the high-bit will be cleared on all characters and single linefeeds will be automatically converted to carriage-return/linefeed sequences.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SetMode](#)

CTelnetClient::GetSecurityInformation Method

```
BOOL GetSecurityInformation(  
    LPSECURITYINFO LpSecurityInfo  
);
```

The **GetSecurityInformation** method returns security protocol, encryption and certificate information about the current client connection.

Parameters

LpSecurityInfo

A pointer to a [SECURITYINFO](#) structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;  
securityInfo.dwSize = sizeof(SECURITYINFO);  
  
if (pClient->GetSecurityInformation(&securityInfo))  
{  
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)  
    {  
        MessageBox(NULL, _T("The connection is not secure"),  
                    _T("Connection"), MB_OK);  
    }  
    else  
    {  
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)  
        {  
            MessageBox(NULL, _T("The connection is secure"),  
                        _T("Connection"), MB_OK);  
        }  
        else  
        {  
            MessageBox(NULL, _T("The server certificate not valid"),  
                        _T("Connection"), MB_OK);  
        }  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [CreateSecurityCredentials](#), [SECURITYINFO](#)

CTelnetClient::GetStatus Method

INT GetStatus();

The **GetStatus** method the current status of the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the client status code. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetStatus** method returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Constant	Description
0	TELNET_STATUS_UNUSED	No connection has been established.
1	TELNET_STATUS_IDLE	The client is current idle and not sending or receiving data.
2	TELNET_STATUS_CONNECT	The client is establishing a connection with the server.
3	TELNET_STATUS_READ	The client is reading data from the server.
4	TELNET_STATUS_WRITE	The client is writing data to the server.
5	TELNET_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[IsBlocking](#), [IsConnected](#), [IsReadable](#), [IsWritable](#)

CTelnetClient::GetTerminalType Method

```
INT GetTerminalType(  
    LPCTSTR lpszTermType,  
    INT nMaxLength  
);  
  
INT GetTerminalType(  
    CString& strTermType  
);
```

The **GetTerminalType** method returns the terminal type for the current client session.

Parameters

lpszTermType

Points to a buffer which the current terminal type is copied into. This buffer should be at least 32 characters in length, including the terminating null character. This argument may also be a **CString** object which will contain the terminal type when the method returns.

nMaxLength

Maximum number of characters that may be copied to the buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is the length of the terminal type name. A value of zero indicates that no terminal type has been specified. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SetTerminalType](#)

CTelnetClient::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[Connect](#), [IsReadable](#), [IsWritable](#), [Read](#), [SetTimeout](#), [Write](#)

CTelnetClient::IsBlocking Method

BOOL IsBlocking();

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking method may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Cancel](#), [GetStatus](#), [IsConnected](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#), [Read](#), [Write](#)

CTelnetClient::IsConnected Method

```
BOOL IsConnected();
```

The **IsConnected** method is used to determine if the client is currently connected to a server.

Parameters

None.

Return Value

If the client is connected to a server, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#)

CTelnetClient::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[CTelnetClient](#), [IsBlocking](#), [IsConnected](#)

CTelnetClient::IsReadable Method

```
BOOL IsReadable(  
    INT nTimeout,  
    LPDWORD lpdwAvail  
);
```

The **IsReadable** method is used to determine if data is available to be read from the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the method returns a non-zero value. If the client cannot read any data, the method returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsConnected](#), [IsInitialized](#), [IsWritable](#), [Write](#)

CTelnetClient::IsThere Method

BOOL IsThere();

The **IsThere** method reports the response of the client to a "Are you there" command to the telnet server.

Parameters

None.

Return Value

The method returns a non-zero value if the server acknowledges a specific control sequence used to determine if a server is responsive. If the server does not respond, the method will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsConnected](#), [IsReadable](#), [IsWritable](#)

CTelnetClient::IsWritable Method

```
BOOL IsWritable(  
    INT nTimeout  
);
```

The **IsWritable** method is used to determine if data can be written to the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the method returns a non-zero value. If the client cannot write any data, the method returns zero.

Remarks

Although this method can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsConnected](#), [IsInitialized](#), [IsReadable](#), [Write](#)

CTelnetClient::Login Method

```
BOOL Login(  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword  
);
```

The **Login** method attempts to authenticate the user and log them in to the current session.

Parameters

lpszUserName

A pointer to a string which specifies the name of the user to authenticate.

lpszPassword

A pointer to a string which specifies the password to be used when authenticating the user. If the user does not require a password, this parameter may be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Login** method is used to authenticate a user, logging them into the server. This method is specifically designed to work with most UNIX based servers, and may work with other servers that use a similar login process. The method works by scanning the data stream for a username prompt and then replying with the specified username. If that is successful, it will then scan for a password prompt and provide the specified password. If no recognized prompt is found, or if the server responds with an error indicating that the username or password is invalid, the method will fail.

If the **Login** method succeeds, the next call to **Read** by the client will return any welcome message to the user. This is typically followed by a command prompt where the user can enter commands to be executed on the server. The data sent by the server during the login process is discarded and not available when the method returns. If the client requires this information, use the **Search** method to automate the login process instead.

Because the **Login** method is designed for UNIX based systems, it may not work with servers running on other operating system platforms such as Windows or VMS. In this case, applications should use the **Search** method to search for the appropriate login prompts in the data stream.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [IsConnected](#), [IsReadable](#), [Read](#), [Search](#)

CTelnetClient::Read Method

```
INT Read(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Read(  
    CString& strBuffer,  
    INT cbBuffer  
);
```

The **Read** method reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be copied. An alternate form of this method allows a **CString** variable to be passed and data read from the socket will be returned in that string.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the method fails, the return value is `TELNET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

When **Read** is called and the client is in non-blocking mode, it is possible that the method will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstntv10.lib`

See Also

[EnableEvents](#), [IsBlocking](#), [IsReadable](#), [IsWritable](#), [RegisterEvent](#), [Write](#)

CTelnetClient::ReadLine Method

```
BOOL ReadLine(  
    LPTSTR lpszBuffer,  
    LPINT lpnLength  
);  
  
BOOL ReadLine(  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);  
  
BOOL ReadLine(  
    CString& strBuffer,  
    INT nMaxLength  
);
```

The **ReadLine** method reads up to a line of data from the server and returns it in a string buffer.

Parameters

lpszBuffer

Pointer to the string buffer that will contain the data when the method returns. The string will be terminated with a null character, and will not contain the end-of-line characters. An alternate form of the method accepts a **CString** argument which will contain the line of text returned by the server.

lpnLength

A pointer to an integer value which specifies the length of the buffer. The value should be initialized to the maximum number of characters that can be copied into the string buffer, including the terminating null character. When the method returns, its value will updated with the actual length of the string.

nMaxLength

An integer value which specifies the maximum length of the buffer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ReadLine** method reads data from the server and copies into a specified string buffer. Unlike the **Read** method which reads arbitrary bytes of data, this method is specifically designed to return a single line of text data in a string. When an end-of-line character sequence is encountered, the method will stop and return the data up to that point. The string buffer is guaranteed to be null-terminated and will not contain the end-of-line characters.

There are some limitations when using **ReadLine**. The method should only be used to read text, never binary data. In particular, the method will discard nulls, linefeed and carriage return control characters. The Unicode version of this method will return a Unicode string, however it does not support reading raw Unicode data from the socket. Any data read from the socket is internally buffered as octets (eight-bit bytes) and converted to Unicode using the **MultiByteToWideChar** function.

This method will force the thread to block until an end-of-line character sequence is processed, the read operation times out or the server closes its end of the socket connection. If this method is

called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, read the data and then restore the socket to asynchronous operation. If another socket operation is attempted while **ReadLine** is blocked waiting for data from the server, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **Read** and **ReadLine** method calls can be intermixed, however be aware that **Read** will consume any data that has already been buffered by the **ReadLine** method and this may have unexpected results.

Unlike the **Read** method, it is possible for data to be returned in the string buffer even if the return value is zero. Applications should check the length of the string to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the return value.

Example

```
CString strBuffer;
BOOL bResult;

do
{
    bResult = pClient->ReadLine(strBuffer);

    if (strBuffer.GetLength() > 0)
    {
        // Process the line of data returned in the string
        // buffer; the string is always null-terminated
    }
} while (bResult);

DWORD dwError = pClient->GetLastError();

if (dwError == ST_ERROR_CONNECTION_CLOSED)
{
    // The server has closed its side of the connection and
    // there is no more data available to be read
}
else if (dwError != 0)
{
    // An error has occurred while reading a line of data
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IsReadable](#), [Read](#), [Write](#), [WriteLine](#)

CTelnetClient::RegisterEvent Method

```
INT RegisterEvent(  
    UINT nEventId,  
    TELNETEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Constant	Description
TELNET_EVENT_CONNECT	The connection to the server has completed.
TELNET_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
TELNET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
TELNET_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
TELNET_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
TELNET_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

LpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **TelnetEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `TELNET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **TelnetEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a `DWORD_PTR` value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstntv10.lib`

See Also

[DisableEvents](#), [EnableEvents](#), [FreezeEvents](#), [TelnetEventProc](#)

CTelnetClient::Search Method

```
BOOL Search(  
    LPCTSTR lpszString  
);  
  
BOOL Search(  
    LPCTSTR lpszString,  
    LPBYTE lpBuffer,  
    LPDWORD lpdwLength  
);  
  
BOOL Search(  
    LPCTSTR lpszString,  
    HGLOBAL* lpBuffer,  
    LPDWORD lpdwLength  
);  
  
BOOL Search(  
    LPCTSTR lpszString,  
    CString& strBuffer  
);
```

The **Search** method searches for a specific character sequence in the data stream and stops reading if the sequence is encountered.

Parameters

lpszString

A pointer to a string which specifies the sequence of characters to search for in the data stream. This parameter cannot be NULL or point to an empty string.

lpBuffer

A pointer to a byte buffer which will contain the output from the server, or a pointer to a global memory handle which will reference the output when the method returns. If the output from the server is not required, this parameter may be NULL.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpBuffer* parameter. If the *lpBuffer* parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual number of bytes of output stored in the buffer. If the *lpBuffer* parameter is NULL, this parameter should also be NULL.

Return Value

If the method succeeds and the character sequences was found in the data stream, the return value is non-zero. If the method fails or a timeout occurs before the sequence is found, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Search** method searches for a character sequence in the data stream and stops reading when it is found. This is useful when the client wants to automate responses to the server, such as logging in a user and executing a command. The method collects the output from the server and stores it in the buffer specified by the *lpBuffer* parameter. When the method returns, the buffer will contain everything sent by the server up to and including the search string.

The *lpBuffer* parameter may be specified in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the a fixed amount of output. In this case, the *lpBuffer* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer. If the server sends more output than can be stored in the buffer, the remaining output will be discarded.

The second method that can be used is have the *lpBuffer* parameter point to a global memory handle which will contain the output when the method returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. This method is preferred if the client application does not have a general idea of how much output will be generated until the search string is found.

Example

```
CString strOutput;
BOOL bResult;

// Search for the login prompt issued by the server

bResult = pClient->Search(T("ogin: "));

// If the Login: prompt was found, then write out the
// username and search for the Password: prompt; note
// that the username, password and command strings are
// terminated with a carriage-return/linefeed sequence
// which the server will see as the user pressing the
// Enter or Return key on the keyboard

if (bResult)
{
    pClient->Write(lpszUserName);
    bResult = pClient->Search(_T("word: "));
}

// If the Password: prompt was found, write out the
// password and then search for the shell prompt;
// the prompt may be different, depending on what
// operating system and shell is being used

if (bResult)
{
    pClient->Write(lpszPassword);
    bResult = Search(hClient, _T("$ "));
}

// If the shell prompt was found, issue the command
// and capture the output into the strOutput string
// and then write it to standard output

if (bResult)
{
    pClient->Write(lpszCommand);

    if (pClient->Search(_T("$ "), strOutput))
        cout << (LPCTSTR)strOutput << endl;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IsBlocking](#), [IsReadable](#), [Login](#), [Read](#)

CTelnetClient::SetLastError Method

```
VOID SetLastError(  
    DWORD dwErrorCode  
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or TELNET_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstntv10.lib

See Also

[GetErrorString](#), [GetLastError](#), [ShowError](#)

CTelnetClient::SetMode Method

```
UINT SetMode(  
    UINT nMode,  
    BOOL bEnable  
);
```

The **SetMode** method sets one or more client modes for the specified session.

Parameters

nMode

The client mode. This value is a combination of one or more flags which determines how the client handles local character echo and character processing. The following values are recognized:

Value	Description
TELNET_MODE_LOCALECHO	The local client is responsible for echoing data entered by the user. By default, this mode is not set which means that the server is responsible for echoing back each character written to it.
TELNET_MODE_BINARY	Data exchanged between the client and server should not be converted or line buffered. If this option is not specified, the high-bit will be cleared on all characters, and single linefeed characters will be converted to carriage-return/linefeed sequences.

bEnable

This boolean flag specifies if the specified mode is to be enabled or disabled.

Return Value

If the method succeeds, the return value is the previous mode. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[GetMode](#)

CTelnetClient::SetTerminalType Method

```
INT SetTerminalType(  
    LPCTSTR lpszTermType  
);
```

The **SetTerminalType** method sets the terminal type for the current client session.

Parameters

lpszTermType

Points to a string which specifies the terminal type.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetTerminalType](#)

CTelnetClient::SetTimeout Method

```
INT SetTimeout(  
    UINT nTimeout  
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[Connect](#), [GetTimeout](#), [IsReadable](#), [IsWritable](#), [Read](#), [Write](#)

CTelnetClient::ShowError Method

```
INT ShowError(  
    LPCTSTR lpszAppTitle,  
    UINT uType,  
    DWORD dwErrorCode  
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A pointer to a string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetErrorString](#), [GetLastError](#), [SetLastError](#)

CTelnetClient::Write Method

```
INT Write(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Write(  
    LPCTSTR lpszBuffer  
    INT cbBuffer  
);
```

The **Write** method sends the specified number of bytes to the server.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server. In an alternate form of the method, the pointer is to a string.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero, unless a pointer to a string buffer is passed as the parameter. In that case, if the value is -1, all of the characters in the string, up to but not including the terminating null character, will be sent to the server.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is TELNET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the next asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableEvents](#), [IsBlocking](#), [IsReadable](#), [IsWritable](#), [Read](#), [RegisterEvent](#)

CTelnetClient::WriteLine Method

```
BOOL WriteLine(  
    LPCTSTR lpszBuffer,  
    LPINT lpnLength  
);
```

The **WriteLine** function sends a line of text to the server, terminated by a carriage-return and linefeed.

Parameters

lpszBuffer

The pointer to a string buffer which contains the data that will be sent to the server. All characters up to, but not including, the terminating null character will be written to the socket. The data will always be terminated with a carriage-return and linefeed control character sequence. If this parameter points to an empty string or NULL pointer, then a only a carriage-return and linefeed are written to the socket.

lpnLength

A pointer to an integer value which will contain the number of characters written to the socket, including the carriage-return and linefeed sequence. If this information is not required, a NULL pointer may be specified.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **WriteLine** method writes a line of text to the server and terminates the line with a carriage-return and linefeed control character sequence. Unlike the **Write** method which writes arbitrary bytes of data to the socket, this method is specifically designed to write a single line of text data from a null-terminated string.

If the *lpszBuffer* string is terminated with a linefeed (LF) or carriage return (CR) character, it will be automatically converted to a standard CRLF end-of-line sequence. Because the string will be sent with a terminating CRLF sequence, the value returned in the *lpnLength* parameter will typically be larger than the original string length (reflecting the additional CR and LF characters), unless the string was already terminated with CRLF.

There are some limitations when using **WriteLine**. This method should only be used to send text, never binary data. In particular, it will discard nulls and append linefeed and carriage return control characters to the data stream. The Unicode version of this method will accept a Unicode string, however it does not support writing raw Unicode data to the socket. Unicode strings will be automatically converted to UTF-8 encoding using the **WideCharToMultiByte** function and then written to the socket as a stream of bytes.

This method will force the thread to block until the complete line of text has been written, the write operation times out or the server aborts the connection. If this method is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, send the data and then restore the socket to asynchronous operation. If another socket operation is attempted while **WriteLine** is blocked sending data to the server, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker

threads to manage each connection.

The **Write** and **WriteLine** methods can be safely intermixed.

Unlike the **Write** function, it is possible for data to have been written to the socket if the return value is zero. For example, if a timeout occurs while the method is waiting to send more data to the server, it will return zero; however, some data may have already been written prior to the error condition. If this is the case, the *lpnLength* argument will specify the number of characters actually written up to that point.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IsWritable](#), [Read](#), [ReadLine](#), [Write](#)

Telnet Protocol Data Structures

- SECURITYCREDENTIALS
- SECURITYINFO

SECURITYCREDENTIALS Structure

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on

	what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that

will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A pointer to a null terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A pointer to a null terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A pointer to a null terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A pointer to a null terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
------------	-------------

CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
----	---

MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
----	--

ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.
------	--

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

A pointer to a null terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used for SSH connections

and should always be NULL when establishing a secure connection using SSL or TLS.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* member should be set to CREDENTIAL_STORE_FILENAME and the *lpzCertStore* member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS#12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the *lpzPassword* member or the library will be unable to access the certificate.

Note that the *lpzUserName* and *lpzPassword* members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

The TLS 1.1 and TLS 1.2 protocols are only supported on Windows 7, Windows Server 2008 R2 and later versions of the platform. If these options are specified and the application is running on Windows XP or Windows Vista, the protocol version will be downgraded to TLS 1.0 for backwards compatibility with those platforms.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

SECURITYINFO Structure

This structure contains information about a secure connection that has been established.

```
typedef struct _SECURITYINFO
{
    DWORD        dwSize;
    DWORD        dwProtocol;
    DWORD        dwCipher;
    DWORD        dwCipherStrength;
    DWORD        dwHash;
    DWORD        dwHashStrength;
    DWORD        dwKeyExchange;
    DWORD        dwCertStatus;
    SYSTEMTIME   stCertIssued;
    SYSTEMTIME   stCertExpires;
    LPCTSTR      lpszCertIssuer;
    LPCTSTR      lpszCertSubject;
    LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to **sizeof(SECURITYINFO)** prior to passing the address of this structure to the function. Note that if this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the protocol that was selected to establish the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The

	<p>correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.</p>
SECURITY_PROTOCOL_TLS10	<p>The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS11	<p>The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS12	<p>The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.</p>

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_CIPHER_RC2	The RC2 block cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
SECURITY_CIPHER_DES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher, using 56-bit

	keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher (also known as the Rijndael cipher) is a fixed block size cipher which use a key size of 128, 192 or 256 bits. This cipher is supported on Windows XP SP3 and later versions of the operating system.
SECURITY_CIPHER_SKIPJACK	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
SECURITY_CIPHER_BLOWFISH	The Blowfish block cipher was selected. This is a variable key length cipher up to 448 bits, using a 64-bit block size.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected. One of the following values will be returned:

Constant	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected. One of the following values will be returned:

--	--

Constant	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography. This key exchange algorithm is only supported on Windows XP SP3 and later versions of the operating system.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server. This member only has meaning for connections using the SSL or TLS protocols. One of the following values will be returned:

Constant	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertIssuer

A pointer to a string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertSubject

A pointer to a string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszFingerprint

A pointer to a string which contains a sequence of hexadecimal values that uniquely identify the server. This member is only used when a connection has been established using the Secure Shell (SSH) protocol.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Unicode: Implemented as Unicode and ANSI versions.

Terminal Emulation Class Library

Emulate an ANSI or DEC VT-220 character mode display terminal.

Reference

- [Class Methods](#)
- [Data Structures](#)
- [Control Sequences](#)
- [Error Codes](#)

Library Information

Class Name	CTerminalEmulator
File Name	CSNVTV10.DLL
Version	10.0.1468.2518
LibID	53152C92-4EA6-4C06-9ED3-50C79C933F01
Import Library	CSNVTV10.LIB
Dependencies	None

Remarks

The Terminal Emulation library provides a comprehensive API for emulating an ANSI or DEC-VT220 terminal, with full support for all standard escape and control sequences, color mapping and other advanced features. The library methods provide both a high level interface for parsing escape sequences and updating a display, as well as lower level primitives for directly managing the virtual display, such as controlling the individual display cells, moving the cursor position and specifying display attributes.

This class can be used in conjunction with the Remote Command, Secure Shell or Telnet Protocol classes to provide terminal emulation services for an application, or it can be used independently. For example, this class could be used to provide emulation services for a program that connects to a device using an RS-232 serial port.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the

file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Terminal Emulation Class Methods

Class	Description
CTerminalEmulator	Constructor which initializes the current instance of the class
~CTerminalEmulator	Destructor which releases resources allocated by the class
Method	Description
AttachHandle	Attach the specified display handle to this instance of the class
Clear	Clear the virtual display
ConvertPos	Convert between pixel and screen coordinates
CopySelectedText	Copy the selected text to the clipboard
Create	Create a new virtual terminal emulation display
DeleteChar	Delete the specified number of characters
DeleteLine	Delete the current line, shifting remaining lines up
DetachHandle	Detach the handle for the current instance of this class
EraseChar	Erase the specified number of characters
EraseLine	Erase the current line
GetAttributes	Return the current display attributes
GetBackColor	Return the current background color
GetCell	Return information about the specified cell in the virtual display
GetCellSize	Return the size of the display character cells
GetColor	Get the current virtual display colors
GetColorMap	Return the virtual display color table
GetCursorPos	Return the current cursor position
GetDisplayInfo	Return information about the virtual display
GetDC	Get the current virtual display device context
GetEmulation	Get the current terminal emulation type
GetFont	Get the current virtual display font
GetForeColor	Return the current foreground color
GetLine	Return a line of text from the virtual display
GetMode	Get the current virtual display mode
GetRect	Return the rectangle for the display window client area
GetScrollPos	Get the display scroll box position
GetSize	Return the current size of the virtual display
GetText	Get the specified block of text from the virtual display

GetWindow	Get the current virtual display window
GetHandle	Return the display handle used by this instance of the class
GetMappedKey	Return the escape sequence for the mapped key
GetScrollRegion	Return the current scrolling region
GetSelectedText	Return the currently selected text
GetTextColor	Return the current text foreground or background color
InsertChar	Insert the specified number of characters
InsertLine	Insert a line, shifting the remaining lines down
IsInitialized	Determine if the class has been successfully initialized
Refresh	Refresh the specified display
Reset	Reset the virtual display
ResetMappedKeys	Reset the mapped key table to default values
Resize	Resize the virtual display
RestoreCursor	Restore the saved cursor position and text attributes
SaveCursor	Save the current cursor position and text attributes
Scroll	Scroll the virtual display
SelectText	Select a region of the virtual display
SetAttributes	Set the current display attributes
SetBackColor	Set the background color for the virtual display
SetBoldColor	Set the bold color for the virtual display
SetCell	Set the value of a character cell in the virtual display
SetColor	Set the virtual display colors
SetColorMap	Set the virtual display color table
SetCursorPos	Set the current cursor position
SetDC	Set the current virtual display device context
SetEmulation	Set the current terminal emulation type
SetFocus	Set the focus on the virtual display
SetFont	Set the current virtual display font
SetForeColor	Set the foreground color for the virtual display
SetMode	Set the current virtual display mode
SetScrollPos	Set the display scroll box position
SetSize	Set the size of the virtual display
SetWindow	Set the current virtual display window
SetMappedKey	Set the escape sequence for the specified key
SetScrollRegion	Set the current scrolling region

SetTextColor	Set the current text foreground or background color
TranslateMappedKey	Translate the keypress to a mapped key escape sequence
Update	Update the window attached to the virtual display
UpdateCaret	Update the display window caret
Write	Write the specified buffer to the virtual display

CTerminalEmulator::CTerminalEmulator Method

`CTerminalEmulator();`

The **CTerminalEmulator** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **NvtInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csnvtv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[~CTerminalEmulator](#), [IsInitialized](#)

CTerminalEmulator::~CTerminalEmulator

`~CTerminalEmulator();`

The **CTerminalEmulator** destructor releases resources allocated by the current instance of the **CTerminalEmulator** object. It also uninitialized the library if there are no other concurrent uses of the class.

Remarks

When a **CTerminalEmulator** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csnvtv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CTerminalEmulator](#)

CTerminalEmulator::AttachHandle Method

```
VOID AttachHandle(  
    HDISPLAY hDisplay  
);
```

The **AttachHandle** method attaches the specified display handle to the current instance of the class.

Parameters

hDisplay

The handle to the virtual display that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a display handle created outside of the class using the SocketTools API. Once the handle is attached to the class, the other class member functions may be used with that virtual display.

The virtual display for the current instance of the class will be destroyed when the new handle is attached to the class object. If you want to prevent the display handle from being released, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the *hDisplay* parameter is presumed to be a valid display handle and no checks are performed to ensure that the handle is valid. Specifying an invalid handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[DetachHandle](#), [GetHandle](#)

CTerminalEmulator::Clear Method

```
BOOL Clear(  
    UINT nMode  
);
```

The **Clear** method clears the specified display, erasing the text and clearing any attributes.

Parameters

nMode

Mode which specifies how the display will be cleared. The following values may be used:

Constant	Description
NVT_CLEAR_EOS	The display is cleared from the current cursor position to the end of the display. The cursor position is not changed.
NVT_CLEAR_TOS	The display is cleared from the beginning of the display to the current cursor position. The cursor position is not changed.
NVT_CLEAR_ALL	The entire display is cleared and the cursor is repositioned to the upper left corner of the display.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[DeleteChar](#), [DeleteLine](#), [EraseChar](#), [EraseLine](#), [Reset](#)

CTerminalEmulator::ConvertPos Method

```
BOOL ConvertPos(  
    INT nMethod,  
    INT xPos,  
    INT yPos,  
    LPPPOINT lppt  
);
```

```
BOOL ConvertPos(  
    INT xPos,  
    INT yPos,  
    LPPPOINT lppt  
);
```

The **ConvertPos** method converts the specified X,Y position and stores the result in the POINT structure provided by the caller.

Parameters

nMethod

An integer value which specifies the conversion method to use. This may be one of the following values:

Constant	Description
NVT_CURSOR_TO_PIXELS	Convert cursor X,Y coordinates to the current window X,Y pixel coordinates. An error is returned if the coordinates are out of bounds for the current display.
NVT_PIXELS_TO_CURSOR	Convert window X,Y pixel coordinates to cursor X,Y coordinates. If the point is outside of the bounds of the display, it is normalized to account for mouse capture.

xPos

An integer value which specifies the X position in the virtual display. This may either be the cursor position or a pixel position, based on the value of the *nMethod* parameter.

yPos

An integer value which specifies the Y position in the virtual display. This may either be the cursor position or a pixel position, based on the value of the *nMethod* parameter.

lppt

A pointer to a POINT structure which will contain the converted coordinates.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

CTerminalEmulator::CopySelectedText Method

BOOL CopySelectedText();

The **CopySelectedText** method copies any selected text to the system clipboard.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csntv10.lib

See Also

[GetSelectedText](#), [SelectText](#)

CTerminalEmulator::Create Method

```
BOOL Create(  
    HWND hWnd,  
    HFONT hFont,  
    UINT nEmulation,  
    UINT nColumns,  
    UINT nRows  
);
```

The **Create** method creates a new virtual terminal emulation display using the specified window and font.

Parameters

hWnd

Handle to the window that will be used to display the virtual terminal.

hFont

Handle to the font that will be used with the virtual terminal. This parameter may be NULL, in which case a default fixed-width font will be used. If a font is specified, it must be fixed-width, otherwise the virtual cursor positioning will be incorrect in some cases.

nEmulation

Identifies the virtual terminal emulation type. The following emulation types are currently supported.

Constant	Description
NVT_EMULATION_NONE	The virtual display does not emulate any specific terminal type, and does not process escape sequences.
NVT_EMULATION_ANSI	The virtual display processes ANSI escape sequences for screen management and cursor positioning. This emulation also supports escape sequences to control the foreground and background color. The default keymap for ANSI function key escape sequences will be selected.
NVT_EMULATION_VT100	The virtual display processes DEC VT-100 escape sequences for screen management and cursor positioning. The default keymap for a DEC VT-100 terminal will be selected.
NVT_EMULATION_VT220	The virtual display processes DEC VT-220 escape sequences for screen management and cursor positioning. This emulation also supports DEC VT-320 escape sequences to control the foreground and background color. The default keymap for a DEC VT-220 terminal will be selected.

nColumns

The maximum number of columns used by the virtual display. This value must be at least 5, and no greater than 255.

nRows

The maximum number of rows used by the virtual display. This value must be at least 5, and no

greater than 127.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero.

Remarks

The default colors for the display is a black background and white foreground. If ANSI terminal emulation is selected, bold characters will be displayed on a white background and blue foreground.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[AttachHandle](#), [DetachHandle](#)

CTerminalEmulator::DeleteChar Method

```
BOOL DeleteChar(  
    INT nChars  
);
```

The **DeleteChar** method deletes the specified number of characters from the display, shifting the characters that follow to the left. The characters are deleted from the current cursor position.

Parameters

nChars

Number of characters to delete from the display.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method does not change the current cursor position. To erase characters from the display without affecting the characters that follow, use the **EraseChar** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csntv10.lib

See Also

[Clear](#), [DeleteLine](#), [EraseChar](#), [EraseLine](#)

CTerminalEmulator::DeleteLine Method

```
BOOL DeleteLine();
```

The **DeleteLine** method deletes the current line, shifting up the lines that follow.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method does not change the current cursor position. To erase a line from the display without affecting the lines that follow, use the **EraseLine** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[Clear](#), [DeleteChar](#), [EraseChar](#), [EraseLine](#)

CTerminalEmulator::DetachHandle Method

```
HDISPLAY DetachHandle();
```

The **DetachHandle** method detaches the display handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the display handle associated with the current instance of the class object. If there is no virtual display associated with that instance of the class, the value `INVALID_DISPLAY` will be returned.

Remarks

This method is used to detach a display handle created by the class for use with the SocketTools API. Once the handle is detached from the class, no other class methods may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csnvtv10.lib`

See Also

[AttachHandle](#), [GetHandle](#)

CTerminalEmulator::EraseChar Method

```
BOOL EraseChar(  
    INT nChars  
);
```

The **EraseChar** method erases the specified number of characters from the display, without affecting the position of the characters that follow. The characters are erased from the current cursor position.

Parameters

nChars

Number of characters to erase from the display.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method does not change the current cursor position. To delete characters from the display and shift the remaining characters to the left, use the **DeleteChar** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[Clear](#), [DeleteChar](#), [DeleteLine](#), [EraseLine](#)

CTerminalEmulator::EraseLine Method

```
BOOL EraseLine(  
    UINT nMode  
);
```

The **EraseLine** method erases the current line without affecting the lines that follow.

Parameters

nMode

Mode which specifies how the line will be erased. The following values may be used:

Value	Description
0	The line is erased from the current cursor position to the end of the line.
1	The line is cleared from the beginning of the line to the current cursor position.
2	The entire line is cleared.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method does not change the current cursor position. To delete a line from the display and shift the remaining lines up, use the **DeleteLine** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Clear](#), [DeleteChar](#), [DeleteLine](#), [EraseChar](#)

CTerminalEmulator::GetAttributes Method

```
UINT GetAttributes();
```

The **GetAttributes** method returns the current display attributes which have been set, either explicitly by the client or as the result of an escape sequence parsed by the emulator.

Parameters

None.

Return Value

If the method succeeds, the return value the current display attributes. If the method fails, the return value is NVT_ERROR.

The following table lists the valid attributes:

Constant	Description
NVT_ATTRIBUTE_NORMAL	Normal, default attributes.
NVT_ATTRIBUTE_REVERSE	Foreground and background cell colors are reversed.
NVT_ATTRIBUTE_BOLD	The character is displayed using a higher intensity color.
NVT_ATTRIBUTE_DIM	The character is displayed using a lower intensity color.
NVT_ATTRIBUTE_UNDERLINE	The character is displayed with an underline.
NVT_ATTRIBUTE_HIDDEN	The character is stored in display memory, but not shown.
NVT_ATTRIBUTE_PROTECT	The character is protected and cannot be cleared.

One or more attributes may be combined using a bitwise Or operator. Certain attributes, such as NVT_ATTRIBUTE_BOLD and NVT_ATTRIBUTE_DIM are mutually exclusive.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[GetCell](#), [SetAttributes](#)

CTerminalEmulator::GetBackColor Method

```
COLORREF GetBackColor();
```

The **GetColor** method returns the background color used by the current display.

Parameters

None.

Return Value

If the method succeeds, the return value is the RGB value of the background color. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[GetForeColor](#), [SetBackColor](#), [SetBoldColor](#), [SetColor](#), [SetBackColor](#)

CTerminalEmulator::GetCell Method

```
BOOL GetCell(  
    INT xPos,  
    INT yPos  
    TCHAR* pChar  
    UINT* pAttributes  
);  
  
BOOL GetCell(  
    TCHAR* pChar  
    UINT* pAttributes  
);
```

The **GetCell** method returns information about the specified character cell in the virtual display.

Parameters

xPos

An integer value which specifies the X cursor position in the display. If this argument is omitted or has the value -1, then the current position in the display is used.

yPos

An integer value which specifies the Y cursor position in the display. If this argument is omitted or has the value -1, then the current position in the display is used.

pChar

A pointer to a buffer that will contain the character in the specified cell when the method returns. If this argument is NULL, then no character is returned.

pAttributes

A pointer to an unsigned integer value which specifies the attributes for the specified character cell. If this argument is NULL, then no attribute information is returned.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero. Failure typically indicates that the X or Y position is invalid, or that a virtual display has not been created.

Remarks

The attribute value returned by the **GetCell** method may be one or more of the following values:

Constant	Description
NVT_ATTRIBUTE_NORMAL	Normal, default attributes.
NVT_ATTRIBUTE_REVERSE	Foreground and background cell colors are reversed.
NVT_ATTRIBUTE_BOLD	The character is displayed using a higher intensity color.
NVT_ATTRIBUTE_DIM	The character is displayed using a lower intensity color.
NVT_ATTRIBUTE_UNDERLINE	The character is displayed with an underline.
NVT_ATTRIBUTE_HIDDEN	The character is stored in display memory, but not shown.
NVT_ATTRIBUTE_PROTECT	The character is protected and cannot be cleared.

One or more attributes may be combined using a bitwise Or operator. Certain attributes, such as

NVT_ATTRIBUTE_BOLD and NVT_ATTRIBUTE_DIM are mutually exclusive.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[GetAttributes](#), [GetCellSize](#), [SetAttributes](#)

CTerminalEmulator::GetCellSize Method

```
BOOL GetCellSize(  
    LPSIZE lpCellSize  
);  
  
BOOL GetCellSize(  
    LPINT lpnWidth  
    LPINT lpnHeight  
);
```

The **GetCellSize** method returns the size of a character cell in pixels.

Parameters

lpCellSize

A pointer to a `SIZE` structure which will contain the size of a character cell in pixels when the method returns.

lpnWidth

A pointer to an integer which will contain the width of a character cell in pixels when the method returns.

lpnHeight

A pointer to an integer which will contain the height of a character cell in pixels when the method returns.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero. Failure typically indicates that the virtual display has not been created.

Remarks

The **GetCellSize** method is used to determine the size of a character cell in the display. This can be useful when the application needs to determine where a display cell is physically located within the virtual display window.

To convert between display and window coordinates, use the **ConvertPos** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csnvtv10.lib`

See Also

[ConvertPos](#), [GetCell](#)

CTerminalEmulator::GetColor Method

```
COLORREF GetColor(  
    UINT nColorIndex,  
    BOOL bForeground  
);
```

The **GetColor** method returns the color used by the virtual display to display text with a specific attribute.

Parameters

nColorIndex

The index into the virtual display color table. It may be one of the following values.

Value	Description
NVT_COLOR_NORMAL	The colors displayed for normal text. These are the default colors used with the display.
NVT_COLOR_REVERSE	The colors displayed for text with the reverse attribute set. This is only used when emulation is enabled.
NVT_COLOR_BOLD	The colors displayed for text with the bold attribute set. This is only used when emulation is enabled.
NVT_COLOR_REVERSEBOLD	The colors displayed for text with the reverse and bold attributes set. This is only used when emulation is enabled.

bForeground

A boolean flag which specifies if the color is a foreground color, used when displaying text, or a background color.

Return Value

If the method succeeds, the return value is the RGB value of the specified color. If the method fails, it returns zero.

Remarks

The default colors for the display is a black background and white foreground. Bold characters are displayed on a white background and blue foreground.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntvt10.lib

See Also

[GetBackColor](#), [GetForeColor](#), [SetBackColor](#), [SetBoldColor](#), [SetColor](#), [SetForeColor](#)

CTerminalEmulator::GetColorMap Method

```
BOOL GetColorMap(  
    COLORREF* lpColor,  
    INT nMaxColors  
);
```

The **GetColorMap** method returns the virtual display color table which determines what RGB values are used to display foreground and background text color attributes.

Parameters

lpColor

A pointer to an array of COLORREF values which will contain the color values currently being used in the virtual display.

nMaxColors

The maximum number of colors which may be stored in the color array. The minimum value for this parameter is 1, and the maximum value is 16.

Return Value

If the method succeeds, the return value is a non-zero value. If the method fails, the return value is zero. Failure indicates that the handle to the virtual display is invalid, the pointer to the color table is NULL or the maximum number of color values is invalid.

Remarks

When the emulator processes an escape sequence that changes the current foreground or background color, the actual RGB color value is determined by looking up the value in the virtual display's color table. The **GetColorMap** method is useful for determining what values are being used when a color attribute is set. The emulator currently supports a maximum of sixteen (16) color values, and the index into the table corresponds to the color as defined by the standard for ANSI terminals:

Index	Color	Default (Hex)	Default (Integer)	Default (RGB)
0	Black	0	0	RGB(0,0,0)
1	Red	000000A0h	160	RGB(160,0,0)
2	Green	0000A000h	40960	RGB(0,160,0)
3	Yellow	0000A0A0h	41120	RGB(160,160,0)
4	Blue	00A00000h	10485760	RGB(0,0,160)
5	Magenta	00A000A0h	10485920	RGB(160,0,160)
6	Cyan	00A0A000h	10526720	RGB(0,160,160)
7	White	00E0E0E0h	14737632	RGB(224,224,224)
8	Gray	00C0C0C0h	12632256	RGB(192,192,192)
9	Light Red	008080FFh	8421631	RGB(255,128,128)
10	Light Green	0090EE90h	9498256	RGB(144,238,144)
11	Light Yellow	00C0FFFFh	12648447	RGB(255,255,192)
12	Light Blue	00E6D8ADh	15128749	RGB(173,216,230)

13	Light Magenta	00FFC0FFh	16761087	RGB(255,192,255)
14	Light Cyan	00FFFFE0h	16777184	RGB(224,255,255)
15	High White	00FFFFFFh	16777215	RGB(255,255,255)

A standard ANSI color terminal supports eight standard colors (0-7). To select a foreground color, you add 30 to the color index and pass that value as a parameter to the SGR (select graphic rendition) escape sequence. To select a background color, you add 40 to the color index. For example, to set the current foreground color to white and the background color to blue, you could send the following escape sequence:

```
ESC [ 37;44 m
```

Note that if you wanted to set the foreground color to a bold version of standard yellow, you would first set the bold attribute, and then use the index value of 3, such as:

```
ESC [ 1;33m
```

GetColorMap is typically used in conjunction with the **SetColorMap** method to load the current color values and then make selective changes to the actual RGB color value that is used when a color attribute is set. Note that changes to the color map will only affect new characters as they are displayed, not any previously displayed characters.

Example

The following example will load the current color table for the virtual display and change the standard white color attribute to use the same value as the high-intensity white:

```
COLORREF rgbColor[16];

if (pDisplay->GetColorMap(rgbColor, 16) )
{
    rgbColor[7] = rgbColor[15];
    pDisplay->SetColorMap(rgbColor, 16);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetColor](#), [GetTextColor](#), [SetColor](#), [SetColorMap](#), [SetTextColor](#)

CTerminalEmulator::GetCursorPos Method

```
BOOL GetCursorPos(  
    LPINT LpnCursorX,  
    LPINT LpnCursorY  
);
```

The **GetCursorPos** method returns the current cursor column and row position on the virtual display.

Parameters

lpnCursorX

Address of the variable that will be set to the column of the current cursor position. If this argument is a NULL pointer, the argument is ignored.

lpnCursorY

Address of the variable that will be set to the row of the current cursor position. If this argument is a NULL pointer, the argument is ignored.

Return Value

If the method succeeds, the return value is a non-zero value. If the handle to the virtual display is invalid, the method will return zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[ConvertPos](#), [SetCursorPos](#)

CTerminalEmulator::GetDC Method

```
HDC GetDC();
```

The **GetDC** returns the device context that has been specified for the virtual display.

Parameters

None.

Return Value

If the method succeeds, the return value is a handle to the device context. If the method fails, or no device context has been specified, the return value is NULL.

Remarks

Normally there is no device context explicitly set for the display. Instead, the device context is dynamically created and released when the virtual display is updated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[SetDC](#), [Update](#)

CTerminalEmulator::GetDisplayInfo Method

```
BOOL GetDisplayInfo(  
    LPNVTDISPLAYINFO Lpvdi  
);
```

The **GetDisplayInfo** method returns information about the specified virtual display.

Parameters

lpvdi

Pointer to a [NVTDISPLAYINFO](#) structure which contains information about the virtual display, including the display window, font, cursor and scrolling position.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[GetCursorPos](#), [GetFont](#), [GetMode](#), [GetRect](#), [GetScrollPos](#)

CTerminalEmulator::GetEmulation Method

```
UINT GetEmulation();
```

The **GetEmulation** method returns the current virtual terminal emulation type.

Parameters

None.

Return Value

If the method succeeds, the return value is the terminal emulation type and may contain one of the following values:

Value	Description
NVT_EMULATION_NONE	The virtual display does not emulate any specific terminal type, and does not process any escape sequences.
NVT_EMULATION_ANSI	The virtual display will process ANSI escape sequences. The default keymap for an ANSI console is loaded.
NVT_EMULATION_VT100	The virtual display will process DEC VT100 escape sequences. The default keymap for a VT100 terminal is loaded.
NVT_EMULATION_VT220	The virtual display will process DEC VT220 escape sequences. The default keymap for a VT220 terminal is loaded.

If the method fails, because an invalid display handle was specified, the return value will be NVT_ERROR.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[Create](#), [Reset](#), [SetEmulation](#)

CTerminalEmulator::GetFont Method

```
HFONT GetFont();
```

The **GetFont** method returns the current font handle being used by the virtual display.

Parameters

None.

Return Value

If the method succeeds, the return value is a handle to the font. If the method fails, it returns NULL.

Remarks

The handle returned by this method should not be released, and the font object should never be directly modified by the application. Functions that return information about the font, such as **GetTextMetrics**, may be safely called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[Create](#), [SetFont](#)

CTerminalEmulator::GetForeColor Method

```
COLORREF GetForeColor();
```

The **GetColor** method returns the foreground color used by the current display.

Parameters

None.

Return Value

If the method succeeds, the return value is the RGB value of the foreground color. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[GetBackColor](#), [SetBackColor](#), [SetBoldColor](#), [SetColor](#), [SetForeColor](#)

CTerminalEmulator::GetHandle Method

```
HDISPLAY GetHandle();
```

The **GetHandle** method returns the display handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the display handle associated with the current instance of the class object. If there is no virtual display associated with that instance of the class, the value `INVALID_DISPLAY` will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csnvtv10.lib`

See Also

[AttachHandle](#), [DetachHandle](#), [IsInitialized](#)

CTerminalEmulator::GetLine Method

```
INT GetLine(  
    INT nRow,  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);
```

```
INT GetLine(  
    INT nRow,  
    CString& strBuffer  
);
```

The **GetLine** method copies a block of text from the specified virtual display into a string buffer.

Parameters

nRow

The row in the virtual display to return the line of text from. The first row in the display is zero. If the value -1 is specified, the row where the cursor is currently located will be used.

lpszBuffer

Pointer to the buffer that the display text will be copied to, terminated with a null character character.

cbBuffer

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the method succeeds, the return value is the number of characters copied into the buffer, not including the null character terminator. If the method fails, it returns zero.

Remarks

The **GetLine** method allows the application to copy the contents of the display at a specific row. Note that the buffer must be large enough to accommodate the text and the null character terminator. Unlike the **GetText** method, any trailing whitespace in the specified row is not copied to the buffer.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetText](#), [Update](#), [Write](#)

CTerminalEmulator::GetMappedKey Method

```
BOOL GetMappedKey(  
    UINT nMappedKey,  
    LPTSTR lpszKeyBuffer,  
    UINT cchKeyBuffer  
);
```

```
BOOL GetMappedKey(  
    UINT nMappedKey,  
    CString& strKeyBuffer  
);
```

The **GetMappedKey** method returns the escape sequence mapped to the specified key.

Parameters

nMappedKey

The key which is an index into the display key mapping table. This defines the values returned by special (method) keys for the current emulation.

lpszKeyBuffer

Address of the buffer which will receive the escape sequence mapped to the specified key. This argument may also be a **CString** object which will contain the mapped key string when the method returns.

cchKeyBuffer

The maximum number of characters that may be copied into the key buffer string, including the terminating null character.

Return Value

If the method succeeds, the return value is non-zero and the escape sequence for the mapped key is copied into the specified buffer. If the key has not been mapped, and there is no default escape sequence defined, then the method will return zero.

Remarks

The **GetMappedKey** method returns the escape sequence that has been mapped to a special key. This method can be used to determine what sequence of characters should be sent in response to a keypress (for example, what sequence should be sent to a server when the user presses the F1 method key). If a sequence has not been explicitly mapped through a call to **SetMappedKey**, the default sequence for the current emulation will be returned.

Note that the current display mode, such as whether or not the emulator is in application mode or not, should be considered when determining which mapped key to use. For example, if the emulator is not in application mode and the user presses the up-arrow key, the sequence mapped to the NVT_UP key should be sent to the server. However, if the emulator is in application mode, the sequence mapped to the NVT_APPUP key should be sent instead. This is automatically handled by the **TranslateMappedKey** method, so it is recommended that it be used when mapping a virtual keypress to the appropriate escape sequence.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetMode](#), [ResetMappedKeys](#), [SetMode](#), [SetMappedKey](#), [TranslateMappedKey](#)

CTerminalEmulator::GetMode Method

```
UINT GetMode();
```

The **GetMode** method returns the current virtual display modes.

Parameters

None.

Return Value

If the method succeeds, the return value is the display mode for the virtual display. If the method fails, it returns zero.

Remarks

The display mode is a combination of one or more flags which determines how the emulator handles automatic line wrapping, caret display, and other methods. The following values are recognized.

Constant	Description
NVT_MODE_AUTOWRAP	The emulator will automatically wrap to the next line when a character is written to the last column on the virtual display.
NVT_MODE_SHOWCARET	The emulator will display a caret when the display receives the focus.
NVT_MODE_BLOCKCARET	The emulator will display a block carat that is the height of the selected font characters. If this mode is not set, the caret is displayed as an underline.
NVT_MODE_BELL	The emulator will beep when a BEL character is processed.
NVT_MODE_CRLF	The cursor will automatically be positioned at the first column when a linefeed character is processed.
NVT_MODE_CRNL	The cursor will automatically advance to the next row when a carriage return character is processed.
NVT_MODE_APPCURSOR	The emulator cursor keys are placed in application mode. This mode changes the default key mappings used when the cursor (arrow) keys are translated. This corresponds to the application mode supported by DEC VT terminals.
NVT_MODE_APPKEYPAD	The emulator keypad keys are placed in application mode. This mode changes the default key mappings used when the keypad keys are translated. This corresponds to the application keypad mode supported by DEC VT terminals.
NVT_MODE_ORIGIN	The emulator is in origin mode. If enabled, the cursor cannot be positioned outside of the current scrolling region. Otherwise, the cursor can be positioned at any valid location on the virtual display.
NVT_MODE_COLOR	The emulator supports the use of escape sequences to change the foreground and background colors. This option is enabled by default if emulating an ANSI console or DEC VT220 terminal.
NVT_MODE_TABOVER	The emulator will clear the character cells between the current

	cursor position and the next tab stop when the HT (horizontal tab) control sequence is processed. By default this mode is disabled, and the cursor is simply positioned at the next tab stop.
NVT_MODE_HSCROLL	The emulator will display a horizontal scroll bar if the number of visible columns are less than that total number of columns in the virtual display. Disabling this mode prevents a horizontal scrollbar from being displayed, regardless of the number of visible columns. By default, this mode is enabled.
NVT_MODE_VSCROLL	The emulator will display a vertical scroll bar if the number of visible rows are less than that total number of rows in the virtual display. Disabling this mode prevents a vertical scrollbar from being displayed, regardless of the number of visible rows. By default, this mode is enabled.
NVT_MODE_NOREFRESH	The emulator will not automatically refresh the window after any change has been made to the virtual display, including changes in the cursor position or display mode. This allows the caller to make a sequence of changes, and then update the display all at one time to prevent a flicker effect.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnavt10.lib

See Also

[GetEmulation](#), [GetDisplayInfo](#), [SetMode](#)

CTerminalEmulator::GetRect Method

```
BOOL GetRect(  
    LPRECT lprc  
);
```

The **GetRect** returns the client rectangle for the virtual display window.

Parameters

lpRect

Pointer to a RECT structure which will contain the client rectangle values when the method returns.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csntv10.lib

See Also

[GetDisplayInfo](#), [GetSize](#), [Resize](#), [SetSize](#)

CTerminalEmulator::GetScrollPos Method

```
INT GetScrollPos(  
    INT nScrollBar  
);
```

The **GetScrollPos** method gets the position of the scroll box for the specified scroll bar.

Parameters

nScrollBar

Specifies the scroll bar to return the position for. This parameter can be one of the following values:

Constant	Description
SB_HORZ	Gets the position of the scroll box in the display's standard horizontal scroll bar.
SB_VERT	Gets the position of the scroll box in the display's standard vertical scroll bar.

Return Value

If the method succeeds, the return value is the position of the scroll box. If the method fails, it returns -1.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[SetScrollPos](#)

CTerminalEmulator::GetScrollRegion Method

```
BOOL GetScrollRegion(  
    LPINT lpnTop,  
    LPINT lpnBottom  
);
```

The **GetScrollRegion** method returns the top and bottom rows of the current scrolling region.

Parameters

lpnTop

Address of the variable that will be set to the top row of the current scrolling region. If no scrolling region has been defined, the value will be 0, the first row in the virtual display. If a NULL pointer is passed as the value, this argument will be ignored.

lpnBottom

Address of the variable that will be set to the bottom row of the current scrolling region. If no scrolling region has been defined, the value will be one less than the maximum number of rows in the virtual display. If a NULL pointer is passed as the value, this argument will be ignored.

Return Value

If the method succeeds, it will return zero. If the handle to the display is invalid, the method will return zero.

Remarks

The **GetScrollRegion** method allows an application to determine the top and bottom rows of the current scrolling region. By default, the scrolling region is the entire virtual display, however this may be changed through a call to **SetScrollRegion** or an ANSI escape sequence. If the display is in origin mode, note that the cursor cannot be positioned outside of the current scrolling region.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[GetMode](#), [SetMode](#), [SetScrollRegion](#)

CTerminalEmulator::GetSelectedText Method

```
INT GetSelectedText(  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);
```

```
INT GetSelectedText(  
    CString& strBuffer  
);
```

The **GetSelectedText** method copies the currently selected text into the specified buffer.

Parameters

lpszBuffer

Pointer to the buffer that the selected text will be copied to, terminated with a null character character. This argument may also be a **CString** object which will contain the selected text when the method returns.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the method succeeds, the return value is the number of characters copied into the buffer, not including the null character terminator. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CopySelectedText](#), [SelectText](#)

CTerminalEmulator::GetSize Method

```
BOOL GetSize(  
    LPSIZE lpSize  
);  
  
BOOL GetSize(  
    LPINT lpnWidth,  
    LPINT lpnHeight  
);
```

The **GetSize** returns the size of the virtual display in columns and rows.

Parameters

lpRect

Pointer to a SIZE structure which will contain the size of the virtual display.

lpnWidth

Pointer to an integer which will contain the number of columns in the virtual display when the method returns. If a NULL pointer is passed as the argument, this value is ignored.

lpnHeight

Pointer to an integer which will contain the number of rows in the virtual display when the method returns. If a NULL pointer is passed as the argument, this value is ignored.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

The **GetSize** method returns the number of columns and rows in the virtual display. To convert this to pixels, use the **GetCellSize** method to determine the size of a character cell and multiply that by the number of columns and/or rows.

Example

```
// Calculate the size of the client area in pixels  
// for the virtual display  
  
INT cxCell = 0, cyCell = 0;  
INT cxDisplay = 0, cyDisplay = 0;  
  
if (pDisplay->GetCellSize(&cxCell, &cyCell))  
{  
    INT nColumns, nRows;  
    if (pDisplay->GetSize(&nColumns, &nRows))  
    {  
        cxDisplay = nColumns * cxCell;  
        cyDisplay = nRows * cyCell;  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[GetCellSize](#), [GetDisplayInfo](#), [GetRect](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CTerminalEmulator::GetText Method

```
INT GetText(  
    INT xPos,  
    INT yPos,  
    LPTSTR lpszBuffer,  
    INT nLength  
);  
  
INT GetText(  
    INT xPos,  
    INT yPos,  
    CString& strBuffer,  
    INT nLength  
);  
  
INT GetText(  
    LPTSTR lpszBuffer,  
    INT nLength  
);  
  
INT GetText(  
    CString& strBuffer,  
    INT nLength  
);
```

The **GetText** method copies a block of text from the specified virtual display into a string buffer.

Parameters

xPos

The starting column where the display text will be copied from. If the method is called where this argument is omitted, the text will be copied from the current cursor position.

yPos

The starting row where the display text will be copied from. If the method is called where this argument is omitted, the text will be copied from the current cursor position.

lpszBuffer

Pointer to the buffer that the display text will be copied to, terminated with a null character character. This argument may also be a **CString** object which will contain the text when the method returns.

nLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator. If a **CString** object is passed to the method, this value may be -1 which specifies that all of the text from the specified position to the last column and row should be copied to the buffer.

Return Value

If the method succeeds, the return value is the number of characters copied into the buffer, not including the null character terminator. If the method fails, it returns zero.

Remarks

The **GetText** method allows the application to copy the contents of the display at a specific location. Note that the buffer must be large enough to accommodate the text and the null

character terminator. To copy an entire row of text in the display, use the **GetLine** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLine](#), [Update](#), [Write](#)

CTerminalEmulator::GetTextColor Method

```
BOOL GetTextColor(  
    COLORREF *LprgbColor,  
    BOOL bForeground  
);
```

The **GetTextColor** method returns the current foreground or background text color.

Parameters

lprgbColor

A pointer to a COLORREF variable which will contain the current foreground or background text color.

bForeground

A boolean value which determines if the foreground or background color is returned. A value of TRUE indicates that the foreground color should be returned, otherwise the background color is returned.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. Failure indicates that the handle to the display is invalid, the display does not support text color attributes, or the pointer to the color value is NULL.

Remarks

This method is used to return the current foreground or background color, as determined by the text attribute. The RGB color value for a color attribute is determined by the virtual display's color table.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[GetColorMap](#), [SetColorMap](#), [SetTextColor](#)

CTerminalEmulator::GetWindow Method

```
HWND GetWindow();
```

The **GetWindow** returns the handle to the window being used by the virtual display.

Parameters

None.

Return Value

If the method succeeds, the return value is a handle to the virtual display window. If the method fails, it returns NULL.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[GetDisplayInfo](#)

CTerminalEmulator::InsertChar Method

```
BOOL InsertChar(  
    INT nChars  
);
```

The **InsertChar** method inserts one or more space characters at the current cursor position.

Parameters

nChars

Number of space characters to insert at the current cursor position.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method does not change the cursor position. Inserting space characters may cause the virtual display to scroll.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[DeleteChar](#), [EraseChar](#), [InsertLine](#)

CTerminalEmulator::InsertLine Method

```
BOOL InsertLine();
```

The **InsertLine** method inserts an empty line at the current row, shifting the remaining lines down.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method does not change the cursor position. Inserting a line may cause the virtual display to scroll.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[DeleteLine](#), [EraseLine](#), [InsertChar](#)

CTerminalEmulator::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[CTerminalEmulator](#)

CTerminalEmulator::Refresh Method

```
BOOL Refresh(  
    BOOL bUpdate  
);
```

The **Refresh** method refreshes the specified virtual display, updating the current scroll position and caret.

Parameters

bUpdate

Boolean flags which specifies if the display window is to be updated. If set, the window client area is invalidated and the virtual display is redrawn.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[UpdateCaret](#), [Update](#), [Write](#)

CTerminalEmulator::Reset Method

```
BOOL Reset(  
    HWND hWnd,  
    HFONT hFont,  
    UINT nColumns,  
    UINT nRows  
);
```

```
BOOL Reset(  
    UINT nColumns,  
    UINT nRows  
);
```

The **Reset** resets the virtual terminal display, using the new window, font, columns and rows. This method should be used when the virtual display must be attached to a different window, or the number of rows or columns must be changed.

Parameters

hWnd

Handle to the window that will be used to display the virtual terminal. This parameter may be NULL, in which case the current window will be used.

hFont

Handle to the font that will be used with the virtual terminal. This parameter may be NULL, in which case the current font will be used. If a font is specified, it must be fixed-width, otherwise the virtual cursor positioning will be incorrect in some cases.

nColumns

The maximum number of columns used by the virtual display. A value of zero specifies that the same number of columns should be used. If a non-zero value is specified, it must be at least 5, and no greater than 255.

nRows

The maximum number of rows used by the virtual display. A value of zero specifies that the same number of columns should be used. If a non-zero value is specified, it must be at least 5, and no greater than 127.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero.

Remarks

This method will clear the display and reset the current text attributes.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[Clear](#), [Create](#), [Refresh](#), [Resize](#), [Update](#)

CTerminalEmulator::ResetMappedKeys Method

BOOL ResetMappedKeys();

The **ResetMappedKeys** method resets all mapped method keys to their default values, based on the current emulation.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the handle to the display is invalid, the method will return zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[GetMappedKey](#), [SetMappedKey](#), [TranslateMappedKey](#)

CTerminalEmulator::Resize Method

```
BOOL Resize(  
    INT cxClient,  
    INT cyClient  
);
```

The **Resize** method resizes the virtual display to the specified width and height.

Parameters

cxClient

New width of the display window in pixels. If this value is zero, the width of the virtual display remains unchanged.

cyClient

New height of the display window in pixels. If this value is zero, the height of the virtual display remains unchanged.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method resizes the virtual display and updates the scrolling information. Typically this method is called when the display window receives a WM_SIZE message, causing the virtual display to match the size of the window's client area.

This method will not change the size of the display window. To change the size of the display window, use the **SetWindowPos** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Create](#), [GetRect](#), [Refresh](#), [Reset](#), [Update](#)

CTerminalEmulator::RestoreCursor Method

```
BOOL RestoreCursor();
```

The **RestoreCursor** method restores the cursor position and text attributes to their previous values.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

Use the **SaveCursor** method to save the current cursor position and text attributes. This method may only be called once for each time that the cursor position is saved.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[GetCursorPos](#), [SaveCursor](#), [SetCursorPos](#), [UpdateCaret](#), [Update](#)

CTerminalEmulator::SaveCursor Method

BOOL SaveCursor();

The **SaveCursor** method saves the current cursor position and text attributes.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

Use the **RestoreCursor** method to restore the cursor position and text attributes to their previous values.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[GetCursorPos](#), [RestoreCursor](#), [SetCursorPos](#), [UpdateCaret](#), [Update](#)

CTerminalEmulator::Scroll Method

```
BOOL Scroll(  
    BOOL bScrollUp  
);
```

The **Scroll** method scrolls the virtual display up or down and updates the scroll box position if necessary.

Parameters

bScrollUp

Boolean flag which specifies if the virtual display is scrolled up or down.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method does not change the current cursor position. The **Refresh** method should be called to update the window attached to the virtual display.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[GetScrollPos](#), [Refresh](#), [SetScrollPos](#)

CTerminalEmulator::SelectText Method

```
BOOL SelectText(  
    LPRECT lprc,  
    DWORD dwOptions  
);
```

The **SelectText** method selects or unselects a region of the virtual display.

Parameters

lprc

A pointer to a region of the display to select. The coordinates must be in display cursor coordinates, not pixels. If this parameter is NULL, any selected text in the display is unselected.

dwOptions

One or more options which specifies how the region will be selected. These options may be combined using a bitwise Or operator. The following values may be used:

Constant	Description
NVT_SELECT_DEFAULT	The default selection option. If there is a region of the display already selected, it will be cleared and the new region is selected.
NVT_SELECT_CLIPBOARD	Copy the selected text to the clipboard. If this option is not specified, the selected text is buffered and may be copied at a later point.
NVT_SELECT_NOREFRESH	The display is not refreshed when the region is selected. This is useful if the application is going to be selecting multiple regions of the display, or combining more than one region, in order to minimize output to the window.
NVT_SELECT_NOBUFFER	Do not buffer the text in the selected region of the display. The display will show any text as being selected, but it will not be available to be copied by the application. This can be useful if the application is going to select multiple regions and combine them.
NVT_SELECT_COMBINE	If there is already a region of the display that has been selected, the new region is combined with the previous region, selecting all of the text.
NVT_SELECT_UNSELECT	Unselect the specified region of the display.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

CTerminalEmulator::SetAttributes Method

```
UINT SetAttributes(  
    UINT nAttributes  
);
```

The **SetAttributes** method sets the current display attributes.

Parameters

nAttributes

An unsigned integer value which specifies the new display attributes. This may be one or more of the following values:

Constant	Description
NVT_ATTRIBUTE_NORMAL	Normal, default attributes.
NVT_ATTRIBUTE_REVERSE	Foreground and background cell colors are reversed.
NVT_ATTRIBUTE_BOLD	The character is displayed using a higher intensity color.
NVT_ATTRIBUTE_DIM	The character is displayed using a lower intensity color.
NVT_ATTRIBUTE_UNDERLINE	The character is displayed with an underline.
NVT_ATTRIBUTE_HIDDEN	The character is stored in display memory, but not shown.
NVT_ATTRIBUTE_PROTECT	The character is protected and cannot be cleared.

Return Value

If the method succeeds, the return value is the previous display attributes. If the method fails, the return value is 0xFFFFFFFF.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[GetAttributes](#), [GetCell](#)

CTerminalEmulator::SetBackColor Method

```
BOOL SetBackColor(  
    COLORREF rgbBackground  
);
```

The **SetBackColor** sets the background color used by the virtual display.

Parameters

rgbBackground

The background color specified as a 32-bit RGB value.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method sets the background color for normal, reverse, bold and reverse-bold text attributes. To set the background color for a specific attribute, use the **SetColor** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[GetBackColor](#), [GetColor](#), [GetForeColor](#), [SetBoldColor](#), [SetColor](#), [SetForeColor](#)

CTerminalEmulator::SetBoldColor Method

```
BOOL SetBoldColor(  
    COLORREF rgbBold  
);
```

The **SetBoldColor** sets the color used by the virtual display to display text with the bold attribute enabled.

Parameters

rgbBold

The bold attribute color specified as a 32-bit RGB value.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method sets the foreground color for the bold and reverse-bold text attributes. To set the color for a specific attribute, use the **SetColor** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[GetColor](#), [SetBackColor](#), [SetColor](#), [SetForeColor](#)

CTerminalEmulator::SetCell Method

```
BOOL SetCell(  
    INT xPos,  
    INT yPos,  
    TCHAR nChar  
    UINT nAttributes  
);
```

```
BOOL SetCell(  
    TCHAR nChar  
    UINT nAttributes  
);
```

The **SetCell** method sets the value of the specified cell in the virtual display.

Parameters

xPos

An integer value which specifies the cell column in the virtual display. If this argument is omitted or a value of -1 is specified, the cell at the current cursor position will be used.

yPos

An integer value which specifies the cell row in the virtual display. If this argument is omitted or a value of -1 is specified, the cell at the current cursor position will be used.

nChar

The character that should be displayed at the specified location in the virtual display

nAttributes

An unsigned integer which specifies the attributes for the character in the virtual display. The attributes may be one or more of the following values, combined using a bitwise Or operator:

Constant	Description
NVT_ATTRIBUTE_NORMAL	Normal, default attributes.
NVT_ATTRIBUTE_REVERSE	Foreground and background cell colors are reversed.
NVT_ATTRIBUTE_BOLD	The character is displayed using a higher intensity color.
NVT_ATTRIBUTE_DIM	The character is displayed using a lower intensity color.
NVT_ATTRIBUTE_UNDERLINE	The character is displayed with an underline.
NVT_ATTRIBUTE_HIDDEN	The character is stored in display memory, but not shown.
NVT_ATTRIBUTE_PROTECT	The character is protected and cannot be cleared.

Note that certain attributes, such as NVT_ATTRIBUTE_BOLD and NVT_ATTRIBUTE_DIM are mutually exclusive.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero. Failure typically indicates that the virtual display has not been created, or that the row and/or column values are outside of the bounds of the display.

Remarks

The **SetCell** method is used to modify a specific character cell in the virtual display, changing both the character and the attributes for that cell. Unlike the higher level methods such as **Write** which process character strings and escape sequences, the **GetCell** and **SetCell** methods allow direct, low-level access to the virtual display in memory. After one or more cells are modified using this function, you should call the **Refresh** method to redraw the virtual display.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[GetAttributes](#), [GetCell](#), [Refresh](#), [SetAttributes](#), [Write](#)

CTerminalEmulator::SetColor Method

```
BOOL SetColor(  
    UINT nColorIndex,  
    COLORREF dwColor,  
    BOOL bForeground  
);
```

The **SetColor** method sets the colors used by the virtual display. Each display has a color table which specifies the foreground and background colors used when displaying text with a specific attribute.

Parameters

nColorIndex

The index into the virtual display color table. It may be one of the following values.

Constant	Description
NVT_COLOR_NORMAL	The colors displayed for normal text. These are the default colors used with the display.
NVT_COLOR_REVERSE	The colors displayed for text with the reverse attribute set. This is only used when emulation is enabled.
NVT_COLOR_BOLD	The colors displayed for text with the bold attribute set. This is only used when emulation is enabled.
NVT_COLOR_REVERSEBOLD	The colors displayed for text with the reverse and bold attributes set. This is only used when emulation is enabled.

dwColor

The RGB color value that will be used.

bForeground

A boolean flag which specifies if the color is a foreground color, used when displaying text, or a background color.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

The default colors for the display is a black background and white foreground. Bold characters are displayed on a white background and blue foreground.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetColor](#), [SetBackColor](#), [SetBoldColor](#), [SetForeColor](#)

CTerminalEmulator::SetColorMap Method

```
BOOL SetColorMap(  
    COLORREF *lpColor,  
    INT nColors  
);
```

The **SetColorMap** method modifies the virtual display color table which determines what RGB values are used to display foreground and background text color attributes.

Parameters

lpColor

A pointer to an array of COLORREF values which specifies the values to be used by the emulator when setting a color attribute. If this value is NULL, the default color table will be loaded.

nColors

The number of colors which are stored in the array. The minimum value for this parameter is 1, and the maximum value is 16.

Return Value

If the method succeeds, the return value is a non-zero value. If the method fails, the return value is zero. Failure indicates that the handle to the virtual display is invalid or the number of color values is invalid.

Remarks

When the emulator processes an escape sequence that changes the current foreground or background color, the actual RGB color value is determined by looking up the value in the virtual display's color table. The **SetColorMap** method is useful for changing what values are being used when a color attribute is set. The emulator currently supports a maximum of sixteen (16) color values, and the index into the table corresponds to the color as defined by the standard for ANSI terminals:

Index	Color	Default (Hex)	Default (Integer)	Default (RGB)
0	Black	0	0	RGB(0,0,0)
1	Red	000000A0h	160	RGB(160,0,0)
2	Green	0000A000h	40960	RGB(0,160,0)
3	Yellow	0000A0A0h	41120	RGB(160,160,0)
4	Blue	00A00000h	10485760	RGB(0,0,160)
5	Magenta	00A000A0h	10485920	RGB(160,0,160)
6	Cyan	00A0A000h	10526720	RGB(0,160,160)
7	White	00E0E0E0h	14737632	RGB(224,224,224)
8	Gray	00C0C0C0h	12632256	RGB(192,192,192)
9	Light Red	008080FFh	8421631	RGB(255,128,128)
10	Light Green	0090EE90h	9498256	RGB(144,238,144)
11	Light Yellow	00C0FFFFh	12648447	RGB(255,255,192)

12	Light Blue	00E6D8ADh	15128749	RGB(173,216,230)
13	Light Magenta	00FFC0FFh	16761087	RGB(255,192,255)
14	Light Cyan	00FFFFE0h	16777184	RGB(224,255,255)
15	High White	00FFFFFFh	16777215	RGB(255,255,255)

A standard ANSI color terminal supports eight standard colors (0-7). To select a foreground color, you add 30 to the color index and pass that value as a parameter to the SGR (select graphic rendition) escape sequence. To select a background color, you add 40 to the color index. For example, to set the current foreground color to white and the background color to blue, you could send the following escape sequence:

```
ESC [ 37;44 m
```

Note that if you wanted to set the foreground color to a bold version of standard yellow, you would first set the bold attribute, and then use the index value of 3, such as:

```
ESC [ 1;33m
```

The **SetColorMap** method is used to modify the actual color displayed by the emulator. For example, if the emulator processes an escape sequence which sets the current foreground color to white, the actual color displayed could be changed to light green. Passing a NULL pointer as the second parameter restores the original color map back to the default values. Note that changes to the color map will only affect new characters as they are displayed, not any previously displayed characters.

Example

The following example will load the current color table for the virtual display and change the standard white color attribute to use the same value as the high-intensity white:

```
COLORREF rgbColor[16];

if (pDisplay->GetColorMap(rgbColor, 16) )
{
    rgbColor[7] = rgbColor[15];
    pDisplay->SetColorMap(rgbColor, 16);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[GetColor](#), [GetColorMap](#), [GetTextColor](#), [SetColor](#), [SetTextColor](#)

CTerminalEmulator::SetCursorPos Method

```
BOOL SetCursorPos(  
    INT nCursorX,  
    INT nCursorY  
);
```

The **SetCursorPos** method sets the current cursor column and row position on the virtual display.

Parameters

nCursorX

New cursor column position. If this value is greater than the maximum number of columns, the current position is set to the last column on the display. The first column on the display is zero.

nCursorY

New cursor row position. If this value is greater than the maximum number of rows, the current position is set to the last row on the display. The first row on the display is zero.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return a value of zero. Typically this would indicate that the virtual display has not been created.

Remarks

The **SetCursorPos** method sets the current cursor position on the virtual display. If the display is in origin mode (a scrolling region has been set), then the cursor row position is bound by the current region.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetCursorPos](#)

CTerminalEmulator::SetDC Method

```
BOOL SetDC(  
    HDC hDC  
);
```

The **SetDC** method sets the device context to be used by the virtual display when updating the window.

Parameters

hDC

Handle to the device context. This parameter may be NULL, any device context that is currently associated with the virtual display will be removed. Note that the application still has the responsibility for deleting the device context, otherwise a handle leak will occur.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

It is not required that the device context be explicitly set by the application. By default, the class will use a device context created using the window attached to the virtual display. If a device context is specified by the application, it must be released when it is no longer needed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[GetDC](#), [Update](#)

CTerminalEmulator::SetEmulation Method

```
BOOL SetEmulation(  
    UINT nEmulation  
);
```

The **SetEmulation** method specifies the type of terminal emulation to be performed by the virtual display.

Parameters

nEmulation

Identifies the virtual terminal emulation type. The following emulation types are currently supported.

Value	Description
NVT_EMULATION_NONE	The virtual display does not emulate any specific terminal type, and does not process any escape sequences.
NVT_EMULATION_ANSI	The virtual display will process ANSI escape sequences. The default keymap for an ANSI console is loaded.
NVT_EMULATION_VT100	The virtual display will process DEC VT100 escape sequences. The default keymap for a VT100 terminal is loaded.
NVT_EMULATION_VT220	The virtual display will process DEC VT220 escape sequences. The default keymap for a VT220 terminal is loaded.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

Changing the emulation type will not affect the current display.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[GetEmulation](#), [GetMode](#), [Reset](#), [SetMode](#)

CTerminalEmulator::SetFocus Method

```
BOOL SetFocus(  
    BOOL bFocus  
);
```

The **SetFocus** method sets or removes the focus from the virtual display. This method should be called when the display window receives or loses focus.

Parameters

bFocus

A boolean flag which specifies that the display should receive or lose focus.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

When the virtual display receives focus, it updates the current cursor position and displays the caret. When the display loses focus, the caret is hidden. This method should be called in response to the display window receiving the WM_SETFOCUS and WM_KILLFOCUS messages.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[UpdateCaret](#)

CTerminalEmulator::SetFont Method

```
BOOL SetFont(  
    HFONT hFont  
);  
  
BOOL SetFont(  
    LPTSTR lpszFontName,  
    INT nFontSize  
);
```

The **SetFont** sets the font that will be used when updating the display. The specified font must be fixed-width, otherwise the virtual cursor positioning will be incorrect in some cases.

Parameters

hFont

Handle to the font that will be used with the virtual terminal. This parameter may be NULL, in which case a default fixed-width font will be used.

lpszFontName

A pointer to a string which specifies the name of the font that will be loaded. If a NULL pointer or an empty string is passed as the value, the default **Terminal** font will be used.

nFontSize

The point size of the font that will be loaded. If this value is zero, a default point size will be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

If the previous font was a default font created by the library as the result of a NULL font handle being passed to a method, it will be released. However, if the previous font was created by the application, the **DeleteObject** method must be called to release it.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[GetFont](#)

CTerminalEmulator::SetForeColor Method

```
BOOL SetForeColor(  
    COLORREF rgbForeground  
);
```

The **SetForeColor** sets the foreground color used by the virtual display.

Parameters

rgbForeground

The foreground color specified as a 32-bit RGB value.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method sets the foreground color for the normal and reverse text attributes. To set the foreground color for a specific attribute, use the **SetColor** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[GetBackColor](#), [GetColor](#), [GetForeColor](#), [SetBackColor](#), [SetBoldColor](#), [SetColor](#)

CTerminalEmulator::SetMappedKey Method

```
BOOL SetMappedKey(  
    UINT nMappedKey,  
    LPCTSTR lpszKeyBuffer  
);
```

The **SetMappedKey** method maps an escape sequence to the specified key.

Parameters

nMappedKey

The key which is an index into the display key mapping table. This defines the values returned by special (method) keys for the current emulation.

lpszKeyBuffer

Pointer to a string which defines the sequence of characters that are to be mapped to the specified key. Passing an empty string or NULL pointer will delete the current sequence mapped to the key and restore the default value, if one has been defined.

Return Value

If the method succeeds, the return value is non-zero and the escape sequence is mapped to the specified key. If the key value is invalid or could not be mapped, then the method will return zero.

Remarks

The **SetMappedKey** method maps an escape sequence to a special key. This method can be used to specify what sequence of characters should be sent in response to a keypress (for example, what sequence should be sent to a server when the user presses the F1 method key). There are a number of default sequences that are mapped to the method and cursor keys, based on the current emulation. Calling this method will override the default sequence for a key, if one has been defined.

The following special keys are defined:

Value	Constant	Description	Value	Constant	Description
0	NVT_F1	F1 method key	26	NVT_UP	Cursor up key
1	NVT_F2	F2 method key	27	NVT_DOWN	Cursor down key
2	NVT_F3	F3 method key	28	NVT_LEFT	Cursor left key
3	NVT_F4	F4 method key	29	NVT_RIGHT	Cursor right key
4	NVT_F5	F5 method key	30	NVT_INSERT	Insert key
5	NVT_F6	F6 method key	31	NVT_DELETE	Delete key
6	NVT_F7	F7 method key	32	NVT_HOME	Home key
7	NVT_F8	F8 method key	33	NVT_END	End key
8	NVT_F9	F9 method key	34	NVT_PGUP	Page up key
9	NVT_F10	F10 method key	35	NVT_PGDN	Page down key
10	NVT_F11	F11 method key	36	NVT_APPUP	Up arrow key
11	NVT_F12	F12 method key	37	NVT_APPDOWN	Down arrow key
12	NVT_SF1	Shift F1 method key	38	NVT_APPLEFT	Left arrow key
13	NVT_SF2	Shift F2 method key	39	NVT_APPRIGHT	Right arrow key
14	NVT_SF3	Shift F3 method key	40	NVT_APPENTER	Keypad enter key
15	NVT_SF4	Shift F4 method key	41	NVT_KEYPAD0	Numeric keypad 0
16	NVT_SF5	Shift F5 method key	42	NVT_KEYPAD1	Numeric keypad 1

17	NVT_SF6	Shift F6 method key	43	NVT_KEYPAD2	Numeric keypad 2
18	NVT_SF7	Shift F7 method key	44	NVT_KEYPAD3	Numeric keypad 3
19	NVT_SF8	Shift F8 method key	45	NVT_KEYPAD4	Numeric keypad 4
20	NVT_SF9	Shift F9 method key	46	NVT_KEYPAD5	Numeric keypad 5
21	NVT_SF10	Shift F10 method key	47	NVT_KEYPAD6	Numeric keypad 6
22	NVT_SF11	Shift F11 method key	48	NVT_KEYPAD7	Numeric keypad 7
23	NVT_SF12	Shift F12 method key	49	NVT_KEYPAD8	Numeric keypad 8
24	NVT_ENTER	Enter key	50	NVT_KEYPAD9	Numeric keypad 9
25	NVT_ERASE	Backspace key			

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetMode](#), [GetMappedKey](#), [ResetMappedKeys](#), [SetMode](#), [TranslateMappedKey](#)

CTerminalEmulator::SetMode Method

```
BOOL SetMode(  
    UINT nMode,  
    BOOL bEnable  
);
```

The **SetMode** method sets one or more display modes for the specified virtual display.

Parameters

nMode

The virtual display mode bitmask. This value is a combination of one or more flags which determines how the emulator handles automatic line wrapping, caret display, and other methods. The following values may be specified.

NVT_MODE_AUTOWRAP	The emulator will automatically wrap to the next line when a character is written to the last column on the virtual display.
NVT_MODE_SHOWCARET	The emulator will display a caret when the display receives the focus.
NVT_MODE_BLOCKCARET	The emulator will display a block carat that is the height of the selected font characters. If this mode is not set, the caret is displayed as an underline.
NVT_MODE_BELL	The emulator will beep when a BEL character is processed.
NVT_MODE_CRLF	The cursor will automatically be positioned at the first column when a linefeed character is processed.
NVT_MODE_CRNL	The cursor will automatically advance to the next row when a carriage return character is processed.
NVT_MODE_APPCURSOR	The emulator cursor keys are placed in application mode. This mode changes the default key mappings used when the cursor (arrow) keys are translated. This corresponds to the application mode supported by DEC VT terminals.
NVT_MODE_APPKEYPAD	The emulator keypad keys are placed in application mode. This mode changes the default key mappings used when the keypad keys are translated. This corresponds to the application keypad mode supported by DEC VT terminals.
NVT_MODE_ORIGIN	The emulator is in origin mode. If enabled, the cursor cannot be positioned outside of the current scrolling region. Otherwise, the cursor can be positioned at any valid location on the virtual display.
NVT_MODE_COLOR	The emulator supports the use of escape sequences to change the foreground and background colors. This option is enabled by default if emulating an ANSI console or DEC VT220 terminal.
NVT_MODE_TABOVER	The emulator will clear the character cells between the current cursor position and the next tab stop when the HT

	(horizontal tab) control sequence is processed. By default this mode is disabled, and the cursor is simply positioned at the next tab stop.
NVT_MODE_HSCROLL	The emulator will display a horizontal scroll bar if the number of visible columns are less than the total number of columns in the virtual display. Disabling this mode prevents a horizontal scroll bar from being displayed, regardless of the number of visible columns. By default, this mode is enabled.
NVT_MODE_VSCROLL	The emulator will display a vertical scroll bar if the number of visible rows are less than the total number of rows in the virtual display. Disabling this mode prevents a vertical scroll bar from being displayed, regardless of the number of visible rows. By default, this mode is enabled.
NVT_MODE_NOREFRESH	The emulator will not automatically refresh the window after any change has been made to the virtual display, including changes in the cursor position or display mode. This allows the caller to make a sequence of changes, and then update the display all at one time to prevent a flicker effect.

bEnable

This boolean flag specifies if the specified mode is to be enabled or disabled.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnavt10.lib

See Also

[Create](#), [GetDisplayInfo](#), [GetMode](#), [SetEmulation](#)

CTerminalEmulator::SetScrollPos Method

```
BOOL SetScrollPos(  
    INT nScrollBar,  
    INT nScrollPos  
);
```

The **SetScrollPos** method sets the position of the scroll box for the specified scroll bar and redraws the scroll bar to reflect the new position of the scroll box.

Parameters

nScrollBar

Specifies the scroll bar to be set. This parameter can be one of the following values:

Value	Description
SB_HORZ	Sets the position of the scroll box in the display's standard horizontal scroll bar.
SB_VERT	Sets the position of the scroll box in the display's standard vertical scroll bar.

nScrollPos

Specifies the new row or column of the scroll box. The position must be within the scrolling range.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method should always be used to change the scroll box position. The **SetScrollPos** method will result in unpredictable behavior if used on the virtual display window.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csntv10.lib

See Also

[GetScrollPos](#), [Scroll](#), [Update](#)

CTerminalEmulator::SetScrollRegion Method

```
BOOL SetScrollRegion(  
    INT nTop,  
    INT nBottom  
);
```

The **SetScrollRegion** method sets the top and bottom rows of the current scrolling region.

Parameters

nTop

The top row of the current scrolling region. If this value is greater than the bottom scrolling region row or the total number of rows in the display, it will be silently adjusted.

nBottom

The bottom row of the current scrolling region. If this value is less than zero or the top scrolling region row, it will be silently adjusted.

Return Value

If the method succeeds, it will return zero. If the handle to the display is invalid, the method will return zero.

Remarks

The **SetScrollRegion** method allows an application to set the current scrolling region for the virtual display. This specifies the region (between the top and bottom rows) in which text will normally scroll. If the display is in origin mode, the cursor cannot be positioned outside of the scrolling region.

The minimum scrolling region that may be defined is two rows. If a scrolling region is specified that is less than two rows, the method will fail and the current scrolling region will remain unchanged. Specifying values of -1 for both arguments will reset the scrolling region to the default values (the full display).

The DEC STBM escape sequence is used to set or clear the scrolling region of the virtual display.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[GetMode](#), [SetMode](#), [SetScrollRegion](#)

CTerminalEmulator::SetSize Method

```
BOOL SetSize(  
    LPSIZE lpSize  
);  
  
BOOL SetSize(  
    INT nColumns,  
    INT nRows  
);
```

The **SetSize** sets the size of the virtual display.

Parameters

lpRect

Pointer to a SIZE structure which specifies the new size of the virtual display. The width should be specified in columns and the height should be specified in rows.

nColumns

The number of columns in the virtual display.

nRows

The number of rows in the virtual display.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csntv10.lib

See Also

[GetDisplayInfo](#), [GetRect](#)

CTerminalEmulator::SetTextColors Method

```
BOOL SetTextColors(  
    COLORREF rgbColor,  
    BOOL bForeground  
);
```

The **SetTextColors** method changes the current foreground or background text color.

Parameters

rgbColor

A color value which specifies the current foreground or background text color. The RGB macro can be used to specify the red, green and blue values for the color.

bForeground

A boolean value which determines if the foreground or background color is changed. A value of TRUE indicates that the foreground color should be changed, otherwise the background color is changed.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. Failure indicates that the handle to the display is invalid or that the current display does not support color text attributes.

Remarks

This method is used to change the current foreground or background color, as determined by the text attribute. Note that changing the current foreground or background text color does not affect the virtual display color table. To change how color attributes are mapped to an RGB color value, use the **SetColorMap** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[GetColorMap](#), [GetTextColor](#), [SetColorMap](#)

CTerminalEmulator::SetWindow Method

```
BOOL SetWindow(  
    HWND hWnd  
);
```

The **SetWindow** method sets the window used by the virtual display.

Parameters

hWnd

Handle to the window that will be used by the virtual display. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method will clear the display and reset the current text attributes.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[Create](#), [GetDC](#), [GetWindow](#), [Reset](#), [SetDC](#)

CTerminalEmulator::TranslateMappedKey Method

```
BOOL TranslateMappedKey(  
    UINT nKey,  
    UINT nFlags,  
    UINT * lpnMappedKey,  
    LPTSTR lpszKeyBuffer,  
    UINT cchBuffer  
);
```

```
BOOL TranslateMappedKey(  
    UINT nKey,  
    UINT nFlags,  
    UINT * lpnMappedKey,  
    CString& strKeyBuffer  
);
```

The **TranslateMappedKey** method translates a virtual key press to an escape sequence based on the current terminal emulation that has been selected.

Parameters

nKey

The virtual key code for the specified key.

nFlags

The scan code, key-transition code, previous key state, and context code for the specified key.

lpnMappedKey

A pointer to an unsigned integer which will contain the index into the keymap table when the method returns. This is the same value used with the **GetMappedKey** and **SetMappedKey** method. If the index into the keymap is not required, this parameter can be NULL.

lpszKeyBuffer

Address of the buffer to receive the escape sequence mapped to the specified key. If the mapped key string is not required, this parameter can be NULL.

cchBuffer

The maximum number of characters that can be copied into the key buffer, including the terminating null character. If the *lpszKeyBuffer* parameter is NULL, this value must be zero.

Return Value

If the virtual key can be mapped to an escape sequence, the method will return a non-zero value. If the key is not mapped, or one of the arguments is invalid, the method will return zero.

Remarks

The **TranslateMappedKey** method allows an application to map a virtual key code to an escape sequence that is appropriate for the type of terminal that is being emulated. For example, it will return the escape sequence for the F1 method key when passed the VK_F1 key value. This method should be called when the WM_KEYDOWN message is processed by an application so that it may send the correct sequence to the server.

This method should only be called in response to a keyboard message such as WM_KEYDOWN. To determine if a specific key has been mapped to an escape sequence, use the **GetMappedKey** method.

Example

```
case WM_KEYDOWN:
/*
 * If the Num Lock key is pressed, then set the terminal into application
 * keypad mode. This will change how the TranslateMappedKey method will
 * translate the keypad keys.
 *
 * Note that the terminal may also be placed into application keypad mode
 * if emulating a DEC VT terminal and the DECNKM escape sequence is sent
 * by the host.
 */
if (wParam == VK_NUMLOCK)
    pDisplay->SetMode(NVT_MODE_APPKEYPAD, !(GetKeyState(VK_NUMLOCK) & 1));
else
{
    BOOL bMapped;
    UINT nMappedKey;
    TCHAR szKey[128];

    bMapped = pDisplay->TranslateMappedKey(wParam, HIWORD(lParam),
                                           &nMappedKey, szKey, 128);

    if (bMapped)
        pTelnet->Write(szKey);
}
break;
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetMode](#), [GetMappedKey](#), [ResetMappedKeys](#), [SetMode](#), [SetMappedKey](#)

CTerminalEmulator::Update Method

BOOL Update();

The **Update** method updates the window attached to the virtual display.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method should be called when the display window receives a WM_PAINT message. If the **SetDC** method has not been called to explicitly set the display device context, this method will acquire one for the display window. In this case, the application should not create device context for the window or call the **BeginPaint** and **EndPaint** methods.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[Refresh](#), [Resize](#), [SetDC](#), [SetScrollPos](#), [SetWindow](#)

CTerminalEmulator::UpdateCaret Method

BOOL UpdateCaret();

The **UpdateCaret** method updates the position of the caret in the display window to the current cursor position.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[SetCursorPos](#), [SetMode](#), [Update](#)

CTerminalEmulator::Write Method

```
BOOL Write(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
BOOL Write(  
    LPCTSTR lpszBuffer,  
    INT cbBuffer  
);  
  
BOOL Write(  
    TCHAR nChar,  
    INT nCount  
);
```

The **Write** method writes the contents of the specified buffer to the virtual display.

Parameters

lpBuffer

Pointer to the buffer which contains the data to be written to the virtual display. This may also specify a string which contains the characters to be written to the display.

cbBuffer

Number of bytes to write to the display. If the buffer is a string, this value may be -1, in which case all characters up to the terminating null character will be written to the display.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it returns zero.

Remarks

This method writes the data at the current cursor location. Control characters are recognized by this method and processed accordingly. If ANSI emulation is enabled, embedded escape sequences will also be parsed and processed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[GetText](#), [Refresh](#), [UpdateCaret](#), [Update](#)

Terminal Emulation Data Structures

- NVTDISPLAYINFO

NVTDISPLAYINFO Structure

This structure is used by the [GetDisplayInfo](#) method to return information about the specified virtual terminal display. No member of this structure should be modified directly by the application.

```
typedef struct _NVTDISPLAYINFO {
    HWND    hWnd;
    HFONT   hFont;
    INT     xPos;
    INT     yPos;
    INT     cxClient;
    INT     cyClient;
    INT     cxChar;
    INT     cyChar;
    INT     nScrollCol;
    INT     nScrollRow;
    INT     nMaxScrollCol;
    INT     nMaxScrollRow;
} NVTDISPLAYINFO, *LPNVTDISPLAYINFO;
```

Members

hWnd

The handle to the terminal emulation display window.

hFont

The handle to the current font.

xPos

The current display x coordinate.

yPos

The current display y coordinate.

cxClient

The width of the client window in pixels.

cyClient

The height of the client window in pixels.

cxChar

The width of the current font in pixels.

cyChar

The height of the current font in pixels.

nScrollCol

The current horizontal scrolling column.

nScrollRow

The current vertical scrolling row.

nMaxScrollCol

The maximum horizontal scrolling column.

nMaxScrollRow

The maximum vertical scrolling row.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Terminal Emulator Control Sequences

Terminal Control Sequences

<ESC>c	Reset display to initial state
<ESC>8	Display alignment test

Cursor Control Sequences

<ESC>D	Move cursor down to next line
<ESC>E	Move cursor to first column and down one line
<ESC>M	Move cursor up one line
<ESC>7	Save cursor position, attributes and colors
<ESC>8	Restore saved cursor position, attributes and colors
<ESC>[nA	Move cursor up <i>n</i> lines
<ESC>[nB	Move cursor down <i>n</i> lines
<ESC>[nC	Move cursor forward <i>n</i> spaces
<ESC>[nD	Move cursor backward <i>n</i> spaces
<ESC>[nE	Move cursor to beginning of line, down <i>n</i> lines
<ESC>[nF	Move cursor to beginning of line, up <i>n</i> lines
<ESC>[xG	Move cursor to column <i>x</i>
<ESC>[y;xH	Move cursor to line <i>y</i> , column <i>x</i>
<ESC>[nI	Move cursor forward <i>n</i> tabstops
<ESC>[nZ	Move cursor backwards <i>n</i> tabstops
<ESC>[na	Move cursor forward <i>n</i> spaces
<ESC>[yd	Move cursor to row <i>y</i>
<ESC>[ne	Move cursor down <i>n</i> lines
<ESC>[y;xf	Move cursor to line <i>y</i> , column <i>x</i>
<ESC>[s	Save cursor position
<ESC>[u	Return to saved cursor position
<ESC>[x`	Move cursor to column <i>x</i>

Attribute and Color Sequence

Select display attributes and color

<i>n</i> Value	Description
0	Reset to default attributes and colors
1	Bold attribute
2	Dim attribute
4	Underline attribute
5	Blink attribute (same as reverse)
7	Reverse attribute
8	Hidden attribute
22	Clear bold attribute
24	Clear underline attribute
25	Clear blink attribute

	27	Clear reverse attribute
	29	Clear color attributes
<ESC>[<i>nm</i>	30	Black foreground
	31	Red foreground
	32	Green foreground
	33	Yellow foreground
	34	Blue foreground
	35	Magenta foreground
	36	Cyan foreground
	37	White foreground
	40	Black background
	41	Red background
	42	Green background
	43	Yellow background
	44	Blue background
	45	Magenta background
	46	Cyan background
	47	White background

Character Set Sequences

<ESC>(A	Assign ISO Latin 1 character set to font bank G0
<ESC>(B	Assign United States ASCII character set to font bank G0
<ESC>(0	Assign graphics character set to font bank G0
<ESC>)A	Assign ISO Latin 1 character set to font bank G1
<ESC>)B	Assign United States ASCII character set to font bank G1
<ESC>)0	Assign graphics character set to font bank G1

Erase Sequences

<ESC>[*n*@ Insert *n* blank spaces
 Erase all or part of the display

<i>n</i> Value	Description
----------------	-------------

<ESC>[<i>n</i> J	0	From current position to end of display
	1	From beginning of display to current position
	2	Erase the entire display

Erase all or part of a line

<i>n</i> Value	Description
----------------	-------------

<ESC>[<i>n</i> K	0	From current position to end of line
	1	From beginning of line to current position
	2	Erase the entire line

<ESC>[*n*L Insert *n* new blank lines

<ESC>[nM Delete *n* lines from current cursor position
<ESC>[nP Delete *n* characters from current cursor position

Scrolling Sequences

<ESC>[nS Scroll display up *n* lines
<ESC>[nT Scroll display down *n* lines
<ESC>[nX Erase *n* characters from the current position
<ESC>[y1;y2r Set scrolling region from lines *y1* to *y2*

Keypad Sequences

<ESC>= Place keypad into applications mode
<ESC>> Place keypad into numeric mode

Emulation Option Sequences

Set emulation option

<i>n</i> Value	Description
----------------	-------------

<ESC>[?nh	1	Enable cursor key application mode
	2	Enable ANSI escape sequences
	5	Reverse foreground and background colors
	6	Enable origin mode
	7	Enable auto-wrap mode
	20	Enable linefeed/newline mode
	25	Display caret
	66	Place keypad in applications mode

Set emulation option

<i>n</i> Value	Description
----------------	-------------

<ESC>[?n1	1	Disable cursor key application mode
	2	Enable VT52 escape sequences
	5	Restore foreground and background colors
	6	Disable origin mode
	7	Disable auto-wrap mode
	20	Disable linefeed/newline mode
	25	Hide caret
	66	Place keypad in numeric mode

Console Escape Sequences

<ESC>[=nA Set the overscan color (ignored)
<ESC>[=n1;n2B Set bell sound (parameters ignored)
<ESC>[=n1;n2C Set the caret size
Set background color intensity

<ESC>[=nD	<i>n</i> Value	Description
	0	Decrease background color intensity

	1	Increase background color intensity
<ESC>[= <i>n</i> E		Set blink vs. bold attribute (ignored)
<ESC>[= <i>n</i> F		Set normal foreground color
<ESC>[= <i>n</i> G		Set normal background color
<ESC>[= <i>n</i> H		Set reverse foreground color
<ESC>[= <i>n</i> I		Set reverse background color
<ESC>[= <i>n</i> J		Set graphics foreground color
		Set graphics background color

	<i>n</i>	Value	Description
	0		Black
	1		Blue
	2		Green
	3		Cyan
	4		Red
	5		Magenta
<ESC>[= <i>n</i> K	6		Brown
	7		White
	8		Gray
	9		Light blue
	10		Light green
	11		Light cyan
	12		Light red
	13		Light magenta
	14		Yellow
	15		High White

Control Character Sequences

<CTL>G	Ring audible bell, if enabled
<CTL>H	Move cursor one character backwards
<CTL>I	Move cursor forward to next tabstop
<CTL>J	Move cursor down to next line
<CTL>M	Move cursor to beginning of line
<CTL>N	Select G1 character set
<CTL>O	Select G0 character set
<CTL>Z	Abort current escape sequence
	Erase and move cursor one character backwards

Text Message Class Library

Send text messages to a mobile communications device using a gateway service.

Reference

- [Class Methods](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

Class Name	CTextMessage
File Name	CSTXTV10.DLL
Version	10.0.1468.2518
LibID	1F65A283-7BC8-4F5B-B739-D211EAF1CA35
Import Library	CSTXTV10.LIB
Dependencies	None

Overview

Short Message Service (SMS) is a text messaging service used by mobile communication devices to exchange brief text messages. Most service providers also provide gateway servers that can be used to send messages to a wireless device on their network using standard email protocols. The **CTextMessage** class provides methods that can be used to determine the provider associated with a specific telephone number and send a text message to the device using the provider's mail gateway.

This library has been designed to assist developers in sending text message notifications as part of their application. For example, it can be used to enable your software to automatically send notifications when a specific event occurs, such as an error condition. This library is not designed to be used with software that will send out a large number of text messages to many users, and there are limitations on the number of messages that may be sent to different phone numbers over a short period of time. Because many recipients must pay a fee for each text message they receive, text messages should only be sent to those who explicitly request them.

Note: This class library only supports service providers in North America and cannot be used to send text messages to mobile devices that use providers outside of the United States and Canada.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Text Message Class Methods

Class	Description
CTextMessage	Constructor which initializes the current instance of the class
~CTextMessage	Destructor which releases resources allocated by the class
Method	Description
DisableRelay	Disable the relaying of messages through another mail server
DisableTrace	Disable logging of network function calls
EnableRelay	Enable the relaying of messages through another mail server
EnableTrace	Enable logging of network function calls to a text file
EnumProviders	Enumerate the available wireless service providers
GetAddress	Return the email address associated with the specified phone number.
GetErrorString	Return a description for the specified error code
GetFirstProvider	Return information about the first supported wireless service provider
GetGateway	Return information about the gateway server for the specified phone number
GetLastError	Return the last error code
GetNextProvider	Return information about the next supported wireless service provider
GetProvider	Return information about the wireless service provider for the specified phone number
GetTimeout	Return the amount of time until an operation times out
IsInitialized	Determine if the class has been successfully initialized
IsRelaying	Determine if relaying through another mail server has been enabled
SendMessage	Send a text message to the specified mobile device
SetLastError	Set the last error code
SetTimeout	Set the amount of time to wait before an operation times out
ShowError	Display a message box with a description of the specified error

CTextMessage::CTextMessage Method

`CTextMessage();`

The **CTextMessage** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **SmsInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstxtv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[~CTextMessage](#), [IsInitialized](#)

CNetMessage::~~CNetMessage

`~CNetMessage();`

The **CNetMessage** destructor releases resources allocated by the current instance of the **CNetMessage** object. It also uninitialized the library if there are no other concurrent uses of the class.

Remarks

When a **CNetMessage** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstxtv10.lib`

See Also

[CNetMessage](#)

CTextMessage::DisableRelay Method

```
VOID DisableRelay();
```

The **DisableRelay** method disables the relaying of text messages through another mail server.

Parameters

None.

Return Value

None.

Remarks

The **DisableRelay** method sets an internal flag that specifies that messages should not be relayed through another mail server. When using the default SMTP service, this means that the library will attempt to send the message directly to the gateway used by the wireless service provider. Relaying is only enabled when the **EnableRelay** method is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableRelay](#), [IsRelaying](#), [SendMessage](#)

CNetMessage::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

See Also

[EnableTrace](#)

CTextMessage::EnableRelay Method

```
BOOL EnableRelay(  
    LPCTSTR lpszHostName,  
    UINT nHostPort,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    DWORD dwOptions,  
);  
  
BOOL EnableRelay(  
    LPCTSTR lpszHostName,  
    UINT nHostPort,  
);  
  
BOOL EnableRelay();
```

The **EnableRelay** method enables the relaying of messages through another mail server.

Parameters

lpszHostName

A pointer to a string that specifies the hostname or IP address of the mail server that the messages will be relayed through. If this parameter is NULL or an empty string, then message relaying will be disabled.

nHostPort

An integer value that specifies the port number that should be used to establish the connection with the mail server. If this value is zero, then the default SMTP port will be used.

lpszUserName

A pointer to a string that specifies the user name that will be used to authenticate the session with the mail server. If the server does not require authentication, this parameter can be NULL or an empty string.

lpszPassword

A pointer to a string that specifies the password that will be used to authenticate the session with the mail server. If the server does not require authentication, this parameter can be NULL or an empty string.

dwOptions

An integer value which specifies one or more options.

Constant	Description
SMS_OPTION_NONE	No additional options specified. This is the default value.
SMS_OPTION_SECURE	This option specifies that SSL/TLS will be used to establish a secure, encrypted connection with the mail server. For some servers, it may be required to connect to them securely and this option will be enabled automatically.

Return Value

If the method succeeds and relaying is enabled, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

When a text message is sent using the SMTP service, the default action is to attempt to connect directly to the wireless service provider's gateway server. However, many residential Internet service providers (ISPs) do not permit their customers to connect to third-party mail servers and will block the outbound connection. Some wireless service providers may also reject messages that originate from residential IP addresses.

To resolve this issue, the developer should allow the user to specify an alternate mail server that will relay the message to the wireless service provider. For residential users, this will typically be the mail server provided by their ISP. For business users, this will usually be their corporate mail server. The **EnableRelay** method is used to specify the connection information for the mail server and the **SendMessage** method will relay messages through that server.

Calling the **EnableRelay** method without any arguments will enable relaying through the mail server specified by a previous call to the method. The **EnableRelay** method must be called at least once with arguments to enable relaying. To temporarily disable relaying, call the **DisableRelay** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableRelay](#), [IsRelaying](#), [SendMessage](#)

CNetMessage::EnableTrace Method

```
BOOL EnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **EnableTrace** method enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace method logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableTrace](#)

CTextMessage::EnumProviders Method

```
INT EnumProviders(  
    LPSMSPROVIDER lpProviders,  
    INT nMaxProviders,  
);
```

The **EnumProviders** method enumerates the supported wireless service providers and populates an array of **SMSPROVIDER** structures.

Parameters

lpProviders

A pointer to an array of **SMSPROVIDER** structures that will be populated with information about each service provider. If this parameter is NULL, the method will return the number of service providers that are known.

nMaxProviders

An integer value which specifies the maximum number of service providers that may be enumerated by this method. If this parameter is zero, the method will return the number of service providers that are known. If the *lpProviders* parameter is NULL, this value should be zero.

Return Value

If the method succeeds, the return value is the number of known service providers. If the method fails, the return value is **SMS_ERROR**. To get extended error information, call **GetLastError**. If the *lpProviders* parameter is not NULL and the *nMaxProviders* parameter indicates the array is not large enough to store all of the provider information, this method will fail with an error indicating that the buffer is too small.

Remarks

The **EnumProviders** method is used to enumerate all of the supported wireless service providers, populating an array of **SMSPROVIDER** structures that contains information about each provider, such as their name, domain, region of the country they service and the maximum message size they will accept. Typically this would be used to update a user interface control such as a listbox or drop-down combobox, enabling a user to select a preferred service provider.

The **GetFirstProvider** and **GetNextProvider** methods offer an alternative way to enumerate the available service providers.

To obtain information about a single service provider, use the **GetProvider** method.

Example

```
SMSPROVIDER smsProviders[MAXPROVIDERS];  
  
INT nProviders = pTextMessage->EnumProviders(smsProviders, MAXPROVIDERS);  
if (nProviders == SMS_ERROR)  
{  
    pTextMessage->ShowError();  
}  
else  
{  
    for (INT nIndex = 0; nIndex < nProviders; nIndex++)  
        pComboBox->AddString(smsProviders[nIndex].szName);  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFirstProvider](#), [GetGateway](#), [GetNextProvider](#), [GetProvider](#), [SMSPROVIDER](#)

CTextMessage::GetAddress Method

```
BOOL GetAddress(  
    LPCTSTR LpszPhoneNumber,  
    LPTSTR LpszAddress,  
    INT nMaxLength,  
);
```

```
BOOL GetAddress(  
    LPCTSTR LpszPhoneNumber,  
    CString& strAddress  
);
```

The **GetAddress** method returns the email address associated with the specified phone number.

Parameters

LpszPhoneNumber

A pointer to a string that specifies the phone number of the mobile device.

LpszAddress

A pointer to a string that will contain the email address associated with the specified phone number when the method returns. If MFC or ATL is being used, an alternate version of this function will return the address in a **CString** type variable.

nMaxLength

The maximum number of characters that may be copied into the string buffer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetAddress** method is used to determine the email address associated with a mobile device. Sending an email message to this address will forward that message to the device as a text message. Internally, this method calls **GetGateway** and returns the address specified in the SMSGATEWAY structure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetGateway](#), [GetProvider](#), [SMSGATEWAY](#)

CErrorMessage::GetErrorString Method

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);
```

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    CString& strDescription  
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstxtv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLastError](#), [SetLastError](#), [ShowError](#)

CTextMessage::GetFirstProvider Method

```
BOOL GetFirstProvider(  
    LPSMSPROVIDER lpProvider  
);
```

The **GetFirstProvider** method returns information about the first supported wireless service provider.

Parameters

lpProvider

A pointer to an [SMSPROVIDER](#) structure that will contain information about the service provider. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **SmsGetLastError**.

Remarks

The **GetFirstProvider** method is used in conjunction with **GetNextProvider** to enumerate all of the supported wireless providers available to the client. These two methods can be used as an alternative to the **EnumProviders** method.

Example

```
SMSPROVIDER smsProvider;  
  
BOOL bResult = pTextMessage->GetFirstProvider(&smsProvider);  
while (bResult)  
{  
    pListBox->AddString(smsProvider.szName);  
    bResult = pTextMessage->GetNextProvider(&smsProvider);  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnumProviders](#), [GetGateway](#), [GetNextProvider](#), [SMSPROVIDER](#)

CTextMessage::GetGateway Method

```
BOOL GetGateway(  
    LPCTSTR lpszPhoneNumber,  
    LPCTSTR lpszProvider,  
    LPSMSGATEWAY lpGateway  
);
```

```
BOOL GetGateway(  
    LPCTSTR lpszPhoneNumber,  
    LPSMSGATEWAY lpGateway  
);
```

The **GetGateway** method returns text message service information for a phone number.

Parameters

lpszPhoneNumber

A pointer to a string which specifies the telephone number that you wish to obtain information about. Any whitespace, punctuation or other non-numeric characters in the string will be ignored. This parameter cannot be NULL.

lpszProvider

A pointer to a string which specifies the preferred service provider for this telephone number. If the preferred service provider is unknown, this parameter can be omitted or specify a NULL pointer and the default provider will be selected.

lpGateway

A pointer to an [SMSGATEWAY](#) structure that will contain information about the text message gateway when the method returns. This includes information such as the name of the provider, the server that will accept text messages for this phone number, and the recipient address that should be used. This parameter cannot be NULL.

Return Value

If the method succeeds and relaying is enabled, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetGateway** method returns information about the service provider and mail gateway for a specific phone number, and can be used to determine if a given phone number is assigned to a mobile device capable of receiving text messages. This is done by sending an query to a server that will check the phone number against a database of known providers and the phone numbers that have been allocated for wireless devices. If the phone number is valid, information will be returned about the provider that is responsible for that number along with information about its text message gateway service.

If the *lpszProvider* parameter is not NULL, this will identify a preferred provider for the phone number specified. In the United States and Canada, most wireless common carriers are required to provide wireless number portability (WNP) which allows a customer to continue to use their current phone number even if they switch to another service provider. This can result in a situation where a specific phone number is shown as allocated to one provider, but in actuality that user has switched to a different provider. For example, a user may have originally purchased a phone and service with AT&T and then later switched to Verizon, but decided to keep their phone number. In this case, if Verizon was not specified as the preferred provider, the library would

attempt to send the message to the AT&T gateway, since that was the original provider who allocated the phone number.

For most applications, the correct way to handle the situation in which a user may have switched to a different service provider is to allow them to select an alternate service provider in your user interface. For example, you could display a drop-down list of available service providers, populated using the **EnumProviders** method. If they select a preferred provider, then you would pass that value to this method. If they do not, then specify a NULL pointer and the default provider will be selected.

This method sends an HTTP query to the server **api.sockettools.com** to obtain information about the phone number and wireless service provider. This requires that the local system can establish a standard network connection over port 80. If the client cannot connect to the server, the method will fail and an appropriate error will be returned. The server imposes a limit on the maximum number of connections that can be established and the maximum number of requests that can be issued per minute. If this method is called multiple times over a short period, the library may also force the application to block briefly. Server responses are cached per session, so calling this method multiple times using the same phone number will not increase the request count.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnumProviders](#), [GetProvider](#), [MSGATEWAY](#)

CErrorMessage::GetLastError Method

```
DWORD GetLastError();  
  
DWORD GetLastError(  
    CString& strDescription  
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError(0)** when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or SMS_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cstxtv10.lib

See Also

[GetErrorString](#), [SetLastError](#), [ShowError](#)

CTextMessage::GetNextProvider Method

```
BOOL GetNextProvider(  
    LPSMSPROVIDER lpProvider,  
);
```

The **GetNextProvider** method returns information about the next supported wireless service provider.

Parameters

lpProvider

A pointer to an [SMSPROVIDER](#) structure that will contain information about the service provider. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the last service provider has been enumerated or the method fails, the return value is zero. To get extended error information, call **SmsGetLastError**.

Remarks

The **GetNextProvider** method is used in conjunction with **GetFirstProvider** to enumerate all of the supported wireless providers available to the client. These two methods can be used as an alternative to the **EnumProviders** method.

Example

```
SMSPROVIDER smsProvider;  
  
BOOL bResult = pTextMessage->GetFirstProvider(&smsProvider);  
while (bResult)  
{  
    pListBox->AddString(smsProvider.szName);  
    bResult = pTextMessage->GetNextProvider(&smsProvider);  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnumProviders](#), [GetGateway](#), [GetNextProvider](#), [SMSPROVIDER](#)

CTextMessage::GetProvider Method

```
BOOL GetProvider(  
    LPCTSTR lpszPhoneNumber,  
    LPSMSPROVIDER lpProvider  
);
```

```
BOOL GetProvider(  
    LPCTSTR lpszPhoneNumber,  
    LPTSTR lpszProvider,  
    INT nMaxLength  
);
```

```
BOOL GetProvider(  
    LPCTSTR lpszPhoneNumber,  
    CString& strProvider  
);
```

The **GetProvider** method returns information about the service provider for the specified phone number.

Parameters

lpszPhoneNumber

A pointer to a string which specifies the telephone number that you wish to obtain information about. Any whitespace, punctuation or other non-numeric characters in the string will be ignored. This parameter cannot be NULL.

lpProvider

A pointer to an [SMS PROVIDER](#) structure that will contain information about the service provider for the specified phone number.

lpszProvider

A pointer to a string that will contain the name of the wireless service provider associated with the specified phone number when the method returns. If MFC or ATL is being used, an alternate version of this function will return the address in a **CString** type variable.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string.

Return Value

If the method succeeds and relaying is enabled, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetProvider** method returns information about the service provider associated with a phone number. This is done by sending an query to a server that will check the phone number against a database of known providers and the phone numbers that have been allocated for wireless devices. If the phone number is valid, information will be returned about the provider that is responsible for that number.

Because most wireless carriers in the United States and Canada must provide for wireless number portability, there is the possibility that the provider information returned may no longer correspond to the telephone number. It is recommended that you provide your end-user with the ability to specify an alternate preferred provider to use when sending the text message. For more

information, refer to the **GetGateway** method.

This method sends an HTTP query to the server **api.sockettools.com** to obtain information about the wireless service provider. This requires that the local system can establish a standard network connection over port 80. If the client cannot connect to the server, the method will fail and an appropriate error will be returned. The server imposes a limit on the maximum number of connections that can be established and the maximum number of requests that can be issued per minute. If this method is called multiple times over a short period, the library may also force the application to block briefly. Server responses are cached per session, so calling this method multiple times using the same phone number will not increase the request count.

For a list of all supported wireless service providers, use the **EnumProviders** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnumProviders](#), [GetGateway](#), [SMS PROVIDER](#)

CTextMessage::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is SMS_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

See Also

[EnableRelay](#), [SendMessage](#), [SetTimeout](#)

CNetMessage::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

See Also

[CNetMessage](#)

CTextMessage::IsRelaying Method

BOOL IsRelaying();

The **IsRelaying** method returns whether or not messages will be relayed through another mail server.

Parameters

None.

Return Value

This method returns a non-zero value if relaying has been enabled, otherwise this method will return zero.

Remarks

For more information about relaying text messages, refer to the **EnableRelay** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

See Also

[DisableRelay](#), [EnableRelay](#), [SendMessage](#)

CTextMessage::SendMessage Method

```
BOOL SendMessage(  
    DWORD dwServiceType,  
    LPCTSTR lpszProvider,  
    LPCTSTR lpszPhoneNumber,  
    LPCTSTR lpszSender,  
    LPCTSTR lpszMessage  
);
```

```
BOOL SendMessage(  
    LPSMSSERVICE lpService,  
    LPSMSMESSAGE lpMessage  
);
```

The **SendMessage** method sends a text message to the specified mobile device.

Parameters

dwServiceType

An unsigned integer value which identifies the type of service being used to send the message. Currently there is only one valid service type, **SMS_SERVICE_SMTP** which sends the message to a mail gateway provided by the wireless service provider. This parameter is included for future expansion when multiple alternative services may be used to send the message.

lpszProvider

A pointer to a string that identifies the wireless service provider that is responsible for the specified phone number. If the service provider is unknown, this parameter can be NULL or an empty string. In that case, the default provider for the phone number will be used.

lpszPhoneNumber

A pointer to a string that specifies the phone number for the mobile device. This can be a standard E.164 formatted number or an unformatted number. Any extraneous whitespace, punctuation or other non-numeric characters in the string will be ignored. This parameter cannot be NULL.

lpszSender

A pointer to a string which identifies the sender of the message and should specify a valid email address. If the recipient replies to the message, the reply will be sent to this address. This parameter cannot be NULL.

lpszMessage

A pointer to a null terminated string that contains the message to be sent to the recipient. In most cases, a message should not exceed 160 characters in length, although some service providers may accept longer messages. If a message exceeds the maximum number of characters accepted by a service provider, the message may be ignored or it may be split into multiple messages.

lpService

A pointer to an [SMSSERVICE](#) structure that identifies the messaging service that will be used to send the text message. The default service sends the message through the mail server gateway for the wireless service provider associated with the recipient's phone number. This parameter cannot be NULL.

lpMessage

A pointer to an [SMSMESSAGE](#) structure that contains information about the message to be sent, including the sender, the recipient and the text message itself. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SendMessage** method is used to send a text message to a mobile device. This API is designed to support multiple methods of sending text messages, with the default method sending the message through a server gateway established by the wireless service provider.

SMS_SERVICE_SMTP

This message service sends the message through the wireless service provider's mail gateway using the SMTP protocol. However, it is important to note that many of these gateways will not accept messages from a client that is connected to them using a residential Internet service provider. If the application is being run on a system that uses a residential provider, that service provider may also block outbound connections to all mail servers other than their own. These anti-spam measures typically require that most end-user applications specify a relay mail server rather than submitting the message directly to the wireless provider's gateway.

Because most wireless carriers in the United States and Canada must provide for wireless number portability, there is the possibility that the provider information returned may no longer correspond to the telephone number. It is recommended that you provide your end-user with the ability to specify an alternate preferred provider to use when sending the text message. For more information, refer to the **GetGateway** method.

This service also sends an HTTP query to the server **api.sockettools.com** to obtain information about the phone number and wireless service provider. This requires that the local system can establish a standard network connection over port 80. If the client cannot connect to the server, the function will fail and an appropriate error will be returned. The server imposes a limit on the maximum number of connections that can be established and the maximum number of requests that can be issued per minute. If this method is called multiple times over a short period, the library may also force the application to block briefly. Server responses are cached per session, so calling this method multiple times using the same phone number will not increase the request count.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnableRelay](#), [GetGateway](#), [GetProvider](#), [SMSMESSAGE](#), [SMSSERVICE](#)

CNetMessage::SetLastError Method

```
VOID SetLastError(  
    DWORD dwErrorCode  
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or SMS_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstxtv10.lib

See Also

[GetErrorString](#), [GetLastError](#), [ShowError](#)

CNetMessage::SetTimeout Method

```
INT SetTimeout(  
    UINT nTimeout  
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the blocking method will fail and return to the caller.

Parameters

nTimeout

The number of seconds until a blocking operation fails. Setting this parameter to zero will use the default timeout period.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is SMS_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cstxtv10.lib

See Also

[GetTimeout](#), [EnableRelay](#), [SendMessage](#)

CErrorMessage::ShowError Method

```
INT ShowError(  
    LPCTSTR lpszAppTitle,  
    UINT uType,  
    DWORD dwErrorCode  
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A pointer to a string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstxtv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetErrorString](#), [GetLastError](#), [SetLastError](#)

Text Message Data Structures

- SMSGATEWAY
- SMSMESSAGE
- SMSPROVIDER
- SMSSERVICE

MSGGATEWAY Structure

This structure contains information about a text message gateway server and the recipient.

```
typedef struct _MSGGATEWAY
{
    INT      nGatewayId;
    INT      nCountryCode;
    INT      nAreaCode;
    INT      nExchange;
    INT      nMessageLength;
    DWORD    dwReserved;
    TCHAR    szProvider[SMS_MAXPROVIDERNAMELEN];
    TCHAR    szDomain[SMS_MAXDOMAINNAMELEN];
    TCHAR    szServer[SMS_MAXMAILSERVERLEN];
    TCHAR    szAddress[SMS_MAXMAILADDRESSLEN];
} SMSCHANNEL, *LPSMSCHANNEL;
```

Members

nGatewayId

An integer value which identifies the gateway record. This value is used internally and an application should not depend on the value not changing over time for a specific telephone number. As new area codes are introduced, the provider database will be updated to reflect these changes and that can result in a change to the gateway ID associated with a specific telephone number.

nCountryCode

An integer value which specifies the ITU country calling code associated with the service provider. Currently this value will always be 1, which is the country code used by North American service providers. If the service provider database is expanded to include additional countries in the future, this value will identify the country of origin.

nAreaCode

An integer value which specifies the Numbering Plan Area (NPA) code, commonly known as the area code. For the United States and Canada, area codes are assigned by the North American Numbering Plan Administration (NANPA). In North America, the area code is digits 1-3 for a 10-digit telephone number. This value, along with the exchange, is used to determine which company provides wireless service for a specific telephone number.

nExchange

An integer value which specifies the exchange area. In North America, the exchange is digits 4-6 for a 10-digit telephone number. This value, along with the area code, is used to determine which company provides wireless service for a specific telephone number.

nMessageLength

An integer value which specifies the maximum number of characters that the service provider will accept for a single text message. If the message exceeds this number of characters, the service provider may reject the message, or it may split the message into multiple messages.

dwReserved

A value reserved for internal use.

szProvider

A pointer to a string which identifies the name of the service provider that is associated with the specified telephone number. Note that this value may not represent the actual company that is

providing the wireless service.

szDomain

A pointer to a string which identifies the gateway domain name used by the service provider to accept text messages for their customer. This domain name is used to determine the actual name of the gateway mail server that is responsible for accepting messages.

szServer

A pointer to a string which identifies the host name or IP address of the mail server used to accept text messages for the specified service provider. In some cases, a provider may have multiple gateway servers and this value will represent the preferred mail server for the domain.

szAddress

A pointer to a string which contains the complete email address that should be used when sending the text message through the gateway mail server. Different service providers can have slightly different rules about how the address is formatted, but it typically is a combination of the telephone number and the gateway domain name.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

SMSMESSAGE Structure

This structure provides information about a text message.

```
typedef struct _SMSMESSAGE
{
    DWORD dwFormat;
    DWORD dwLength;
    DWORD dwFlags;
    DWORD dwReserved;
    LPCTSTR lpszProvider;
    LPCTSTR lpszPhoneNumber;
    LPCTSTR lpszSender;
    LPCTSTR lpszMessage;
} SMSMESSAGE, *LPSMSMESSAGE;
```

Members

dwFormat

An integer value which specifies the format of the message message. This member is included for future use where a service provider supports multiple message formats based on different versions of the protocol. The default value for this member is SMS_FORMAT_TEXT.

dwLength

An integer value which specifies the length of the message. If this member is zero, the length will be automatically calculated based on the length of the *lpszMessage* text that is terminated by a null character. If this value is larger than the actual length of the message text, it will be ignored.

dwFlags

An integer value which specifies one or more message options.

Constant	Description
SMS_MESSAGE_DEFAULT	The default value used with standard text messages.
SMS_MESSAGE_URGENT	The text message should be flagged as urgent. For messages that are sent through a mail gateway, this will set the header to indicate that it is a high priority message. Note that service providers handle urgent messages differently and some may ignore the message priority.

dwReserved

Reserved for future use. This value should always be zero.

lpszProvider

A pointer to a null terminated string which specifies the name of the preferred wireless service provider responsible for handling the message. If this member is NULL or an empty string, the default provider assigned to the recipient's phone number will be used. This structure member is only used with SMS_SERVICE_SMTP messages and is ignored for other message services.

lpszPhoneNumber

A pointer to a null terminated string which specifies the recipient's phone number. This can be a standard E.164 formatted number or an unformatted number. Any extraneous whitespace, punctuation or other non-numeric characters in the string will be ignored. This structure member cannot be NULL.

lpszSender

A pointer to a null terminated string which identifies the sender of the message. For SMS_SERVICE_SMTP messages, this string should be a valid email address. For other services, this string may specify a phone number or shortcode. This structure member cannot be NULL.

lpszMessage

A pointer to a null terminated string that contains the message to be sent to the recipient. In most cases, a message should not exceed 160 characters in length, although some service providers may accept longer messages. If a message exceeds the maximum number of characters accepted by a service provider, the message may be ignored or it may be split into multiple messages.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SendMessage](#), [SMSSERVICE](#)

SMSPROVIDER Structure

This structure contains information about a wireless service provider.

```
typedef struct _SMSPROVIDER
{
    INT        nProviderId;
    INT        nCountryCode;
    INT        nRegionCode;
    INT        nMessageLength;
    DWORD      dwFlags;
    DWORD      dwReserved;
    TCHAR      szGuid[SMS_MAXPROVIDERGUIDLEN];
    TCHAR      szName[SMS_MAXPROVIDERNAMELEN];
    TCHAR      szCompany[SMS_MAXCOMPANYNAMELEN];
    TCHAR      szDomain[SMS_MAXDOMAINNAMELEN];
} SMSPROVIDER, *LPSMSPROVIDER;
```

Members

nProviderId

An integer value which identifies the provider record. This value is used internally and an application should not depend on the value not changing over time for a specific service provider. To uniquely identify a provider, use the *szGuid* member of the structure, which is guaranteed not to change as providers are added and removed from the database.

nCountryCode

An integer value which specifies the ITU country calling code associated with the service provider. Currently this value will always be 1, which is the country code used by North American service providers. If the service provider database is expanded to include additional countries in the future, this value will identify the country of origin.

nRegionCode

An integer value which identifies the region that the service provider covers. In North America, each region consists of multiple states and/or provinces. If a provider services multiple regions, this will identify the primary region where they provide coverage.

Constant	Description
SMS_REGION_NATIONAL 0	All regions. This region code is used for service providers that have national coverage and do not exclusively provide service within one or more specific geographical regions.
SMS_REGION_NORTH_EAST_ATLANTIC 1	Northeastern Atlantic region which includes Connecticut, Maine, Massachusetts, New Hampshire, New Brunswick, Newfoundland, Nova Scotia, Rhode Island and Vermont.
SMS_REGION_MIDDLE_ATLANTIC 2	Middle Atlantic region which includes New Jersey, New York, Delaware, District of Columbia, Maryland, Pennsylvania, Quebec, Virginia and West Virginia.
SMS_REGION_EAST_NORTH_CENTRAL 3	Northeastern central region which includes Illinois, Indiana, Michigan, Ohio and

	Wisconsin.
SMS_REGION_SOUTH_ATLANTIC 4	Southern Atlantic region which includes Florida, Georgia, North Carolina and South Carolina.
SMS_REGION_EAST_SOUTH_CENTRAL 5	Southeastern central region which includes Alabama, Kentucky, Mississippi and Tennessee.
SMS_REGION_WEST_NORTH_CENTRAL 6	Northwestern central region which includes Iowa, Kansas, Manitoba, Minnesota, Missouri, Nebraska, North Dakota, Ontario and South Dakota.
SMS_REGION_WEST_SOUTH_CENTRAL 7	Southwestern central region which includes Arkansas, Louisiana, Oklahoma and Texas.
SMS_REGION_MOUNTAIN 8	Mountain region which includes Alberta, Arizona, Colorado, Idaho, Montana, Nevada, New Mexico, Northwest Territories, Saskatchewan, Utah and Wyoming.
SMS_REGION_PACIFIC 9	Pacific region which includes Alaska, British Columbia, California, Hawaii, Oregon, Washington and Yukon.

nMessageLength

An integer value which specifies the maximum number of characters that the service provider will accept for a single text message. If the message exceeds this number of characters, the service provider may reject the message, or it may split the message into multiple messages.

dwFlags

An unsigned integer value which specifies one or more flags that provides additional information about the service provider. This value is constructed by using a bitwise operator with any of the following constants:

Constant	Description
SMS_PROVIDER_DEFAULT 0	A standard service provider. Typically this means that customers have a service contract for their mobile device and pay monthly access and service charges.
SMS_PROVIDER_PREPAID 1	A service provider that offers pre-paid calling cards or fixed month-to-month payments that do not require long-term service contracts.

dwReserved

A value reserved for internal use.

szGuid

A pointer to a string which uniquely identifies the service provider. The string is in a standard format used for globally unique identifiers (GUIDs) and is guaranteed to not change for the service provider it has been assigned to.

szName

A pointer to a string which specifies the name of service provider. Note that this value may not represent the actual company that is providing the wireless service.

szCompany

A pointer to a string which specifies the name of the company associated with the service provider. This may be the same as name of the service provider itself or it may be the name of a parent company that owns the service provider.

szDomain

A pointer to a string which identifies the gateway domain name used by the service provider to accept text messages for their customer. This domain name is used to determine the actual name of the gateway mail server that is responsible for accepting messages.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

SMSSERVICE Structure

This structure provides information about the service used to send a text message.

```
typedef struct _SMSSERVICE
{
    DWORD dwServiceType;
    DWORD dwAuthType;
    DWORD dwVersion;
    DWORD dwTimeout;
    DWORD dwOptions;
    DWORD dwReserved;
    LPCTSTR lpszResource;
    LPCTSTR lpszAccount;
    LPCTSTR lpszUserName;
    LPCTSTR lpszPassword;
} SMSSERVICE, *LPSMSSERVICE;
```

Members

dwServiceType

An integer value which identifies the type of service that will be used to send the message. This member can be one of the following values:

Constant	Description
SMS_SERVICE_SMTP	The text message will be sent through the mail gateway for the specified service provider. This service uses SMTP to submit the message for delivery, either directly to the server provider's mail gateway server or through a relay server. This is the default service type.

dwAuthType

An integer value which identifies the type of authentication used with the service. This member can be one of the following values:

Constant	Description
SMS_AUTH_DEFAULT	The default authentication method for this service type should be used. Most applications should use this value unless a service type provides multiple authentication methods.
SMS_AUTH_USERNAME	The service requires authentication using a username and password. This value can be used with an SMTP service that requires user authentication and is typically needed when using a mail server relay. For the SMS_SERVICE_SMTP service, this is the default authentication method.

dwVersion

An integer value which identifies the interface version for the service being used. This member is included for future use where a service may support multiple versions of their interface and should normally be set to the value SMS_VERSION_DEFAULT.

dwTimeout

An integer value which specifies the amount of time in seconds that a function will wait for a

response from the service. If this value is zero, a default timeout period of 20 seconds will be used. If the service does not respond within this time period, the function will fail.

dwOptions

An integer value which specifies one or more options.

Constant	Description
SMS_OPTION_NONE	No additional options for the service.
SMS_OPTION_SECURE	This option specifies that SSL/TLS will be used to establish a secure, encrypted connection with the service. For some services, it may be required to connect to them securely and this option will be enabled automatically.

dwReserved

Reserved for future use. This value should always be zero.

lpszResource

A pointer to a null terminated string that specifies a resource for the service. Typically this will be either a fully qualified domain name or a URL. For gateways using SMTP, this string should identify the mail server. An alternate port number can also be specified by appending it to the hostname, separated by a colon. For example, **smtp.company.com:587** would connect to the server on port 587. If you are specifying an IPv6 address with an alternate port number, the address must be enclosed in brackets. For services where a domain name or resource URL is not required, this member will be ignored and can be NULL.

lpszAccount

A pointer to a null terminated string that specifies an account name or identifier. Some service providers may require a unique account name or a token (application ID) in conjunction with other credentials. This member is not used with mail gateways and ignored if the service type is SMS_SERVICE_SMTP. If no account name is required for session authentication, this member can be NULL.

lpszUserName

A pointer to a null terminated string that specifies a user name to authenticate the session. If the authentication type is SMS_AUTH_USERNAME this member must specify a valid user name. If no authentication is required, this member may be NULL. Note that some service providers may use terminology other than "username" with their documentation; this member will always specify the first of a pair of authentication tokens.

lpszPassword

A pointer to a null terminated string that specifies the password used to authenticate the session. If the authentication type is SMS_AUTH_USERNAME this member must specify a valid password. If no authentication is required, this member may be NULL. Note that some service providers may use terminology other than "password" with their documentation; this member will always specify the second of a pair of authentication tokens.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SendMessage](#), [SMSMESSAGE](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

CWebStorage Application Storage Class Library

The CWebStorage class provides data storage for applications.

Reference

- [Class Methods](#)
- [Constants](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

Class Name	CWebStorage
File Name	CSWEBV10.DLL
Version	10.0.1468.2518
LibID	62AB4693-FFA1-4A6F-BC44-FDC113EB2C68
Import Library	CSWEBV10.LIB
Dependencies	None

Overview

The CWebStorage class enables an application to store and manage data remotely. These methods use secure services provided by SocketTools API servers and do not require third-party party APIs or accounts with other cloud service providers.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

CWebStorage Class Methods

Class	Description
CWebStorage	Constructor which initializes the current instance of the class
~CWebStorage	Destructor which releases resources allocated by the class
Method	Description
Cancel	Cancel a storage operation that is currently in progress
CloseStorage	Close the storage container and release resources allocated for the client session
CompareFile	Compare the contents of a stored object with a local file
CompareData	Compare the contents of a stored object with a memory buffer
CompareText	Compare the contents of a stored object with a string
CopyObject	Copy a storage object to a new location and optionally rename the object label
DeleteObject	Delete an existing storage object
DownloadFile	Download the contents of a stored object to a local file
DisableTrace	Disable logging of method calls to the trace log file
EnableTrace	Enable logging of method calls to a log file
EnumObjects	Enumerate all storage objects that match the specified label or content type
GetAccountId	Return the current web services account identifier
GetData	Download the contents of a storage object to a memory buffer
GetErrorString	Return a description for the specified error code
GetFile	Download the contents of a storage object to a local file
GetFirstApplication	Return information about the first registered application
GetFirstObject	Return information about the first object that matches search criteria
GetHandle	Return the storage handle used by this instance of the class
GetLastError	Return the last error code
GetNextApplication	Return information about the next registered application
GetNextObject	Return information about the next object that matches search criteria
GetObjectInformation	Retrieve the metadata for the specified storage object
GetObjectSize	Return the size of the specified storage object
GetStorageQuota	Return quota limits assigned to your storage account
GetStorageId	Return the returns the current storage container ID
GetTimeout	Get the number of seconds until a storage operation times out
GetTransferStatus	Return status information about the progress of a data transfer
IsBlocking	Determine if the session waiting for a response from the storage server
IsConnected	Determine if the client is connected to the storage server

IsInitialized	Determine if the class has been successfully initialized
MoveObject	Move a storage object to a new location and optionally rename the object label
OpenStorage	Open a storage container and return a handle for the client session
PutData	Upload the contents of a memory buffer and return information about the new object
PutFile	Upload the contents of a local file and return information about the new object
RegisterAppId	Register a new application identifier used to store and retrieve data
RegisterEvent	Register an event handler to receive notifications for the session
RenameObject	Change the label associated with a storage object
ResetStorage	Resets the application storage container and deletes all stored objects
SetLastError	Set the last error code
SetTimeout	Set the number of seconds until a storage operation times out
ShowError	Display a message box with a description of the specified error
UnregisterAppId	Unregister the application identifier and delete all associated storage objects
UnregisterEvent	Unregister an event handler and stop receiving notifications for the session
UploadFile	Upload the contents of a local file
ValidateAppId	Validate the specified application identifier
ValidateLabel	Check the specified string to ensure it is a valid object label

CWebStorage::CWebStorage

`CWebStorage()`;

The **CWebStorage** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only method on the development system and cannot be redistributed.

The constructor calls the **WebInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors.

You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cswebv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[~CWebStorage](#), [IsInitialized](#), [WebInitialize](#)

CWebStorage::~~CWebStorage

`~CWebStorage();`

The **CWebStorage** destructor releases resources allocated by the current instance of the **CWebStorage** object. It also uninitialized the library if there are no other concurrent uses of the class.

Remarks

When a **CWebStorage** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cswebv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CWebStorage](#), [IsConnected](#), [IsInitialized](#)

CWebStorage::Cancel Method

```
BOOL Cancel();
```

The **Cancel** method cancels the current data transfer in progress.

Parameters

None.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **Cancel** method will cancel the current data transfer and abort the connection to the storage server. This method will fail if an active data transfer (either an upload or download) is not in progress.

This would typically be used within an event handler to cancel an operation as the contents of an object are being read or written. There is no mechanism to resume a canceled data transfer and this method should only be used when absolutely necessary.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

See Also

[GetTransferStatus](#), [RegisterEvent](#)

CWebStorage::CloseStorage Method

```
BOOL CloseStorage();
```

The **CloseStorage** method closes the open storage container.

Parameters

None.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **CloseStorage** method must be called after all operations using the storage container have completed. The access token granted to the application will be released and the memory allocated for the session cache will be freed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

See Also

[GetHandle](#), [OpenStorage](#)

CWebStorage::CompareData Method

```
BOOL CompareData(  
    LPCTSTR lpszObjectLabel,  
    LPCVOID lpvBuffer,  
    DWORD dwLength  
);
```

The **CompareData** method compares the contents of a stored object with the data provided by the caller.

Parameters

lpszObjectLabel

A pointer to a null terminated string which specifies the label of the object to be compared.

lpvBuffer

A pointer to a buffer that contains the data to be compared against the storage object.

dwLength

An unsigned integer which specifies the length of the data buffer to be compared.

Return Value

If the method succeeds, the return value is a non-zero and the data matches the contents of the stored object. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **CompareData** method performs a binary comparison of the data in the specified buffer with the contents of the storage object on the server. The *dwLength* parameter must match the size of the stored object exactly, or this method will fail. Partial comparisons are not supported by this method.

If you wish to compare the contents of a text object, it is recommended that you use **CompareText**. This method ensures that Unicode text is compared correctly.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CompareFile](#), [CompareText](#), [GetObjectInformation](#)

CWebStorage::CompareFile Method

```
BOOL CompareFile(  
    LPCTSTR lpszObjectLabel,  
    LPCTSTR lpszLocalFile  
);
```

The **CompareFile** method compares the contents of a stored object with a local file.

Parameters

lpszObjectLabel

A pointer to a null terminated string which specifies the label of the object to be compared.

lpszLocalFile

A pointer to a null terminated string that specifies the name of the local file. If a path is not specified, the file will be created in the current working directory.

Return Value

If the method succeeds, the return value is a non-zero and the contents of the file matches the stored object. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **CompareFile** method performs a binary comparison of the contents of a local file with a stored object on the server. The contents of the file must be identical to the contents of the stored object or the method will fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CompareData](#), [CompareText](#), [GetObjectInformation](#)

CWebStorage::CompareText Method

```
BOOL CompareText(  
    LPCTSTR lpszObjectLabel,  
    LPCTSTR lpszObjectText,  
    INT cchObjectText  
);
```

The **CompareText** method compares the contents of a stored object with the string provided by the caller.

Parameters

lpszObjectLabel

A pointer to a null terminated string which specifies the label of the object to be compared.

lpszObjectText

A pointer to a null terminated string which contains the text to be compared with the stored object.

cchObjectText

The number of characters in the *lpszObjectText* string to be compared. This value may be -1, in which case the string length will be determined by counting the number of characters up to the terminating null.

Return Value

If the method succeeds, the return value is a non-zero and the data matches the contents of the stored object. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **CompareText** method performs a text comparison of the characters in the string with the contents of the storage object on the server. The string length must match the amount of text in the stored object exactly, or this function will fail. Partial comparisons are not supported by this function.

Unicode strings will be automatically converted from UTF-16 to UTF-8 encoding in the same way the **PutData** method does when storing text objects.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CompareFile](#), [CompareData](#), [GetData](#), [GetObjectInformation](#), [PutData](#)

CWebStorage::CopyObject Method

```
BOOL CopyObject(  
    LPCTSTR lpszOldLabel,  
    LPCTSTR lpszNewLabel,  
    DWORD dwStorageType,  
    LPWEB_STORAGE_OBJECT lpObject  
);
```

The **CopyObject** method creates a copy of an existing storage object using a new label.

Parameters

lpszOldLabel

A pointer to a null terminated string which specifies the label of the existing storage object to be copied. This parameter must specify a valid object label and cannot be a NULL pointer or an empty string.

lpszNewLabel

A pointer to a null terminated string which specifies the name of the new storage object that will be created. This parameter may be NULL or point to an empty string, in which case the label name is not changed. In this case, the **dwStorageType** parameter cannot be WEB_STORAGE_DEFAULT.

dwStorageType

An integer value that identifies the storage container type. If this parameter is omitted, the value WEB_STORAGE_DEFAULT will be used. One of the following values can be specified:

Constant	Description
WEB_STORAGE_DEFAULT (0)	The default storage type. If this value is specified, the new object will be created using the same storage type as the original storage object.
WEB_STORAGE_GLOBAL (1)	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN (2)	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
WEB_STORAGE_MACHINE (3)	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is

	reformatted.
WEB_STORAGE_USER (4)	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain information about the new storage object. If this information is not required, this parameter may be omitted or a NULL pointer.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **CopyObject** method is used to create a copy of an existing storage object. It may be used to duplicate an object with a different label, or it may be used to copy the object to a new storage container type. For example, it can copy an object originally created using WEB_STORAGE_USER to a new object stored using WEB_STORAGE_MACHINE.

Copied objects are assigned their own unique ID and are not linked to one another. Any subsequent changes made to the original object will not affect the copied object. Attempting to copy an object to itself or another existing object will result in an error.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DeleteObject](#), [MoveObject](#), [RenameObject](#)

CWebStorage::DeleteObject Method

```
BOOL DeleteObject(  
    LPCTSTR lpszObjectLabel  
);
```

The **DeleteObject** method deletes an object from the storage container.

Parameters

lpszObjectLabel

A pointer to a null terminated string which specifies the label of the object to be deleted.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **DeleteObject** method permanently deletes the storage object and its associated data from the server. Deleted objects cannot be recovered by the application. To remove all objects stored in the container, use the **ResetStorage** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CopyObject](#), [MoveObject](#), [RenameObject](#), [ResetStorage](#)

CWebStorage::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

See Also

[EnableTrace](#)

CWebStorage::DownloadFile Method

```
BOOL DownloadFile(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszObjectLabel,  
    LPWEB_STORAGE_OBJECT LpObject  
);
```

```
BOOL DownloadFile(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszObjectLabel,  
    LPCTSTR lpszAppId,  
    DWORD dwStorageType,  
    DWORD dwTimeout,  
    LPWEB_STORAGE_OBJECT LpObject,  
    WEBEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

The **DownloadFile** method downloads the contents of a storage object and copies it to a local file.

Parameters

lpszLocalFile

A pointer to a null terminated string that specifies the name of the local file that will be created or overwritten with the contents of the storage object. If a path is not specified, the file will be created in the current working directory.

lpszObjectLabel

A pointer to a null terminated string that specifies the label of the object that should be retrieved from the server.

lpszAppId

A pointer to null terminated string which specifies the application ID for the storage container. The application ID is a string that uniquely identifies the application and can only contain letters, numbers, the period and the underscore character. If this parameter is NULL or an empty string, the default identifier **SocketTools.Storage.Default** will be used.

dwStorageType

An integer value that identifies the storage container type. If this parameter is omitted, WEB_STORAGE_GLOBAL will be used as the default. One of the following values can be specified:

Constant	Description
WEB_STORAGE_GLOBAL (1)	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN (2)	Local domain storage. Objects stored using this storage type are only available to users in the same local domain,

	as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
WEB_STORAGE_MACHINE (3)	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER (4)	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

dwTimeout

An integer value that specifies a timeout period in seconds. If this value is zero, a default timeout period will be used.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain additional information about the object that has been downloaded. This parameter may be omitted or NULL if the information is not required.

lpEventProc

Specifies the procedure-instance address of the application defined callback method. For more information about the callback method, see the description of the **WebEventProc** callback method. If this parameter is omitted or NULL, no callback method will be invoked during the data transfer.

dwParam

A user-defined integer value that is passed to the callback method specified by *lpEventProc*. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **DownloadFile** method downloads the contents of the storage object and stores it in a local file. Unlike the **GetFile** method, it is not required that you explicitly open a storage container using the **OpenStorage** method prior to calling this method.

Additional metadata about the object will be returned in the [WEB_STORAGE_OBJECT](#) structure provided by the caller, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents.

If you are downloading a large object and want your application to receive progress updates during the data transfer, provide a pointer to a static callback function as the *lpEventProc* parameter. That function will receive event notifications as the data is being downloaded.

Example

```
WEB_STORAGE_OBJECT webObject;

// Download the object from global storage to a local file
if (pStorage->DownloadFile(lpszObjectLabel, lpszLocalFile, &webObject))
{
    // The object was downloaded, display the metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The object could not be retrieved, display the error
    CString strError;

    pStorage->GetLastError(strError);
    _tprintf(_T("Unable to retrieve \"%s\" (%s)\n"), lpszObjectLabel,
(LPCTSTR)strError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFile](#), [PutFile](#), [UploadFile](#),

CWebStorage::EnableTrace Method

```
BOOL EnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **EnableTrace** method enables the logging of socket method calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All method calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those method calls which fail are recorded in the trace file.
TRACE_WARNING	Only those method calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All methods calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All method calls in the current process are logged, rather than only those methods in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket method call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace method logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableTrace](#)

CWebStorage::EnumObjects Method

```
BOOL EnumObjects(  
    LPCTSTR lpszMatchLabel,  
    LPCTSTR lpszContentType,  
    LPWEB_STORAGE_OBJECT& lpObjects,  
    DWORD& dwObjects  
);  
  
DWORD EnumObjects(  
    LPCTSTR lpszMatchLabel,  
    LPCTSTR lpszContentType  
);
```

The **EnumObjects** method enumerates all storage objects that match the specified label or content type.

Parameters

lpszMatchLabel

A pointer to a null terminated string which specifies the value to match against the object labels in the container. The string may contain wildcard characters similar to those use with the Windows filesystem. A "?" character matches any single character, and "*" matches any number of characters in the label. If this value is a NULL pointer or an empty string, all objects in the container will be matched.

lpszContentType

A pointer to a null terminated string which specifies the content type of the objects to be enumerated. If this value is a NULL pointer or an empty string, the content type is ignored and all matching objects are returned.

lpObjects

A pointer to an array of [WEB_STORAGE_OBJECT](#) structures that will contain information about the enumerated objects. If this parameter is NULL, then no object information is returned.

lpdwObjects

A pointer to an unsigned integer that will contain the number of objects enumerated by this method. If the *lpObjects* parameter points to an array of [WEB_STORAGE_OBJECT](#) structures, this parameter must be initialized to the maximum size of the array being passed to the method. If the *lpObjects* parameter is NULL, this value must be initialized to zero, and when the method returns it will contain the number of matching objects.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **EnumObjects** method can be used to enumerate the number of objects that match a given label, content type or both. If a content type is specified, it must be a valid MIME media content type designated using the type/subtype nomenclature. For example, "text/plain" or "image/jpeg". An invalid MIME type will cause the method to fail.

This method can be used in two ways. If the *lpObjects* parameter is NULL and the value passed by reference in the *lpdwObjects* parameter is initialized to zero, the method will return the number of matching objects in *lpdwObjects*. This can be used to dynamically determine the size of the

lpObjects array instead of declaring a fixed-size array in your application.

If the *lpObjects* parameter is not NULL, then the value referenced by *lpdwObjects* must be initialized to the maximum size of the array that *lpObjects* points to. It is important to note that if either parameter is not initialized correctly, it can result in memory corruption and/or an unhandled exception.

There is an alternative version of **EnumObjects** which only returns a count of objects that match the specified content type and/or object label name, without returning information about the objects. This can be used to determine how many objects match the given criteria prior to requesting the object metadata.

The **GetFirstObject** and **GetNextObject** methods can be used to iterate through all matching storage objects without allocating memory to store all of the matching objects. Using **GetFirstObject** and **GetNextObject** is more efficient when the container contains a large number of objects that match the specified label and/or content type.

Example

```
DWORD dwObjects = 0;
LPWEB_STORAGE_OBJECT lpObjects = NULL;

if (pStorage->EnumObjects(_T("*.pdf"), NULL, lpObjects, dwObjects))
{
    // Print information about each object
    for (DWORD dwIndex = 0; dwIndex < dwObjects; dwIndex++)
    {
        _tprintf(_T("Object: %s\n"), lpObjects[dwIndex].szObjectId);
        _tprintf(_T("Label: %s\n"), lpObjects[dwIndex].szLabel);
        _tprintf(_T("Size: %lu\n"), lpObjects[dwIndex].dwObjectSize);
        _tprintf(_T("Digest: %s\n"), lpObjects[dwIndex].szDigest);
        _tprintf(_T("Content: %s\n"), lpObjects[dwIndex].szContent);

        if (dwObjects > 1 && dwIndex < dwObjects - 1)
            _tprintf(_T("\n"));
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFirstObject](#), [GetNextObject](#), [GetObjectInformation](#)

CWebStorage::GetAccountId Method

```
BOOL GetAccountId(  
    LPCTSTR lpszAccountId,  
    INT nMaxLength  
);  
  
BOOL GetAccountId(  
    CString& strAccountId  
);
```

The **GetAccountId** method returns the web services account ID associated with the current session.

Parameters

lpszAccountId

A pointer to a null-terminated string that will contain the account ID when the method returns. This parameter cannot be NULL and must be at least 35 characters in length. An alternate form of the method accepts a **CString** variable which will contain the account ID.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the *lpszAccountId* string parameter, including the terminating null character.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The account ID is a string that uniquely identifies the web services account that is associated with the session. The account ID corresponds with your product serial number and runtime license key, but it is not identical to either of those values.



If you are using an evaluation license, the account ID is temporary and only valid during the evaluation period. After the evaluation period has expired, the account ID is revoked and objects stored using this ID will be deleted. It is not recommended that you store critical application data using an evaluation license.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetStorageId](#)

CWebStorage::GetErrorString Method

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);  
  
INT GetErrorString(  
    DWORD dwErrorCode,  
    CString& strDescription  
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLastError](#), [SetLastError](#), [ShowError](#)

CWebStorage::GetData Method

```
BOOL GetData(  
    LPCTSTR lpszObjectLabel,  
    LPBYTE lpBuffer,  
    LPDWORD lpdwLength,  
    LPWEB_STORAGE_OBJECT lpObject,  
);
```

```
BOOL GetData(  
    LPCTSTR lpszObjectLabel,  
    HGLOBAL * lphgblBuffer,  
    LPDWORD lpdwLength,  
    LPWEB_STORAGE_OBJECT lpObject,  
);
```

```
BOOL GetData(  
    LPCTSTR lpszObjectLabel,  
    LPTSTR lpszBuffer,  
    INT nMaxLength,  
    LPWEB_STORAGE_OBJECT lpObject,  
);
```

```
BOOL GetData(  
    LPCTSTR lpszObjectLabel,  
    CString& strBuffer,  
    LPWEB_STORAGE_OBJECT lpObject,  
);
```

The **GetData** method retrieves the contents of a storage object and copies it to the memory buffer that is provided. Additional information about the object is optionally returned to the caller.

Parameters

lpszObjectLabel

A pointer to a null terminated string that specifies the label of the object that should be retrieved from the server.

lpBuffer

A pointer to a byte array which will contain the object's data. When this version of the method is called, the buffer must be pre-allocated by the application and large enough to store all of the data or the method will fail. The *lpdwLength* parameter must be initialized with the maximum size of the byte array, and will be updated with the actual number of bytes copied into the buffer when the method returns.

lphgblBuffer

A pointer to a HGLOBAL memory handle. When this version of the method returns, the handle will reference a block of memory allocated by the **GlobalAlloc** function and the *lpdwLength* parameter will contain the number of bytes copied. To obtain a pointer to the data, the application must call the **GlobalLock** function. It is the responsibility of the application to free the global memory handle when it is no longer needed.

lpszBuffer

A pointer to a string buffer which will contain the object's text when the method returns. When this version of the method is called, the string will be populated with the object contents and terminated with a null character. The *nMaxLength* parameter must specify the maximum

number of characters that can be copied into the string, including the terminating null. This version of the function should only be used with text and must never be used with binary data.

strBuffer

A **CString** which will contain the object's text when the method returns. This version of the method requires the application be compiled with MFC or ATL support. This version of the function should only be used with text and must never be used with binary data.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpBuffer* parameter. If the *lpBuffer* parameter points to a global memory handle, the length value should be initialized to zero. When the method returns, this value will be updated with the actual length of the file that was downloaded.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the *lpzBuffer* string buffer provided by the caller. This value must be large enough to contain the entire contents of the object, including the terminating null character.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain additional information about the object. This parameter may be omitted or NULL if the information is not required.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **GetData** method is used to retrieve a stored object from the server and copy it into a local buffer. The method may be used in one of several ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the object. In this case, the *lpBuffer* parameter will point to the buffer that was allocated and the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is to provide a pointer to an HGLOBAL global memory handle which will contain the file data when the method returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the method must be freed by the application, otherwise a memory leak will occur. See the example code below.

The third method is to provide a string buffer which will contain the contents of a text object. The *nMaxLength* parameter must specify a size large enough to contain the entire object text, including a terminating null character. If the **PutData** method was previously used to store UTF-16 encoded text, you must use a byte array to retrieve that data. This method only recognizes UTF-8 encoded text and considers UTF-16 encoded text to be binary data. This version of the method should never be used to retrieve data from an object that does not contain text.

If you wish to retrieve the contents of an object and store it in a file, use the **GetFile** method.

Additional metadata about the object can be returned in a [WEB_STORAGE_OBJECT](#) structure provided by the caller, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents.

Example

```

HGLOBAL hgblBuffer = (HGLOBAL)NULL;
DWORD dwLength = 0;
LPCTSTR lpszObjectLabel = _T("MyAppData");

// Open the default global storage container
if (!pStorage->OpenStorage(WEB_STORAGE_GLOBAL))
{
    _tprintf(_T("Unable to open global storage\n"));
    return;
}

// Return the file data into block of global memory allocated by
// the GlobalAlloc method; the handle to this memory will be
// returned in the hgblBuffer parameter
if (pStorage->GetData(lpszObjectLabel, &hgblBuffer, &dwLength))
{
    // Lock the global memory handle, returning a pointer to the
    // resource data
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
else
{
    // The GetData method failed
    _tprintf(_T("Unable to retrieve object \"%s\"\n"), lpszObjectLabel);
}

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFile](#), [GetObjectSize](#), [GetObjectTime](#), [PutFile](#), [PutData](#)

CWebStorage::GetFile Method

```
BOOL GetFile(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszObjectLabel,  
    LPWEB_STORAGE_OBJECT lpObject  
);
```

The **GetFile** method downloads the contents of a storage object and copies it to a local file.

Parameters

lpszLocalFile

A pointer to a null terminated string that specifies the name of the local file that will be created or overwritten with the contents of the storage object. If a path is not specified, the file will be created in the current working directory.

lpszObjectLabel

A pointer to a null terminated string that specifies the label of the object that should be retrieved from the server.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain additional information about the object. This parameter may be omitted or NULL if the information is not required.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

If you are downloading a large object and want your application to receive progress updates during the data transfer, use the **RegisterEvent** method and provide a pointer to a static callback function that will receive event notifications.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cswebv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DownloadFile](#), [GetData](#), [PutFile](#), [PutData](#), [RegisterEvent](#), [UploadFile](#)

GetFirstApplication Method

```
BOOL GetFirstApplication(  
    LPWEB_STORAGE_APPLICATION LpAppInfo  
);
```

The **GetFirstApplication** method returns information about the first registered application for the current storage account.

Parameters

LpAppInfo

A pointer to a [WEB_STORAGE_APPLICATION](#) structure that will contain information about the registered application when the function returns. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetFirstApplication** method returns information about the first registered application. It is used in conjunction with the **GetNextApplication** method to enumerate all of the registered application IDs associated with your storage account.

The cached application ID information used by this method is shared by the entire process. Attempting to enumerate all registered application IDs in multiple threads at the same time can yield unexpected results. It is recommended that multi-threaded clients use a critical section to ensure that only a single thread is enumerating the AppIDs at any one time.

Example

```
WEB_STORAGE_APPLICATION webApp;  
  
if (pStorage->GetFirstApplication(&webApp))  
{  
    do  
    {  
        // Print information for each registered application  
        _tprintf(_T("AppId: %s\n"), webApp.szAppId);  
        _tprintf(_T("Key: %s\n"), webApp.szApiKey);  
        _tprintf(_T("LUID: %s\n"), webApp.szLuid);  
        _tprintf(_T("Tokens: %lu\n"), webApp.dwTokens);  
        _tprintf(_T("\n"));  
    }  
    while (pStorage->GetNextApplication(&webApp));  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetNextApplication](#), [RegisterAppId](#), [UnregisterAppId](#)

GetFirstObject Method

```
BOOL GetFirstObject(  
    LPWEB_STORAGE_OBJECT LpObject  
);  
  
BOOL GetFirstObject(  
    LPCTSTR LpszMatchLabel,  
    LPWEB_STORAGE_OBJECT LpObject  
);  
  
BOOL GetFirstObject(  
    LPCTSTR LpszMatchLabel,  
    LPCTSTR LpszContentType,  
    LPWEB_STORAGE_OBJECT LpObject  
);
```

The **GetFirstObject** method returns information about the first storage object that matches the specified label or content type.

Parameters

LpszMatchLabel

A pointer to a null terminated string which specifies the value to match against the object labels in the container. The string may contain wildcard characters similar to those use with the Windows filesystem. A "?" character matches any single character, and "*" matches any number of characters in the label. If this parameter is omitted, a NULL pointer or an empty string, all objects in the container will be matched.

LpszContentType

A pointer to a null terminated string which specifies the content type of the objects to be enumerated. If this parameter is omitted, a NULL pointer or an empty string, the content type is ignored and all matching objects are returned. If a content type is specified, it must be a valid MIME media content type designated using the type/subtype nomenclature.

LpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain information about the storage object when the function returns. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **GetFirstObject** method returns information about the first object that matches a given label, content type or both. It is used in conjunction with the **GetNextObject** method to enumerate all of the matching objects in the storage container.

This method provides an alternative to the **EnumObjects** method, which populates an array of [WEB_STORAGE_OBJECT](#) structures. In some cases, iterating over each object in a loop may be preferred to preallocating memory for an array to store every matching object. Using **GetFirstObject** and **GetNextObject** is more efficient when the container contains a large number of objects that match the specified label and/or content type.

You cannot intermix calls between **GetFirstObject** and **EnumObjects**. The **EnumObjects** method will reset the internal object cache for the client session and subsequent calls to **GetNextObject**

will fail.

Example

```
WEB_STORAGE_OBJECT webObject;

if (pStorage->GetFirstObject(&webObject))
{
    do
    {
        // Print information about each object
        _tprintf(_T("Object: %s\n"), webObject.szObjectId);
        _tprintf(_T("Label: %s\n"), webObject.szLabel);
        _tprintf(_T("Size: %lu\n"), webObject.dwObjectSize);
        _tprintf(_T("Digest: %s\n"), webObject.szDigest);
        _tprintf(_T("Content: %s\n"), webObject.szContent);
        _tprintf(_T("\n"));
    }
    while (pStorage->GetNextObject(&webObject));
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnumObjects](#), [GetNextObject](#)

CWebStorage::GetHandle Method

```
HSTORAGE GetHandle();
```

The **GetHandle** method returns the storage handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_HANDLE` will be returned.

Remarks

This method is used to obtain the storage handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cswebv10.lib`

See Also

[IsInitialized](#)

CWebStorage::GetLastError Method

```
DWORD GetLastError();  
  
DWORD GetLastError(  
    CString& strDescription  
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Methods set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError(0)** when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Method failure is typically indicated by a return value such as FALSE, NULL, INVALID_HANDLE or WEBAPI_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswebv10.lib

See Also

[GetErrorString](#), [SetLastError](#), [ShowError](#)

GetNextApplication Function

```
BOOL GetNextApplication(  
    LPWEB_STORAGE_APPLICATION LpAppInfo  
);
```

The **GetNextApplication** method returns information about the next registered application for the current storage account.

Parameters

LpAppInfo

A pointer to a [WEB_STORAGE_APPLICATION](#) structure that will contain information about the registered application when the function returns. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetNextApplication** method returns information about the next registered application after initially calling the **GetFirstApplication** method. It is used to enumerate all of the registered application IDs associated with your storage account.

The cached application ID information used by this method is shared by the entire process. Attempting to enumerate all registered application IDs in multiple threads at the same time can yield unexpected results. It is recommended that multi-threaded clients use a critical section to ensure that only a single thread is enumerating the AppIDs at any one time.

Example

```
WEB_STORAGE_APPLICATION webApp;  
  
if (pStorage->GetFirstApplication(&webApp))  
{  
    do  
    {  
        // Print information for each registered application  
        _tprintf(_T("AppId: %s\n"), webApp.szAppId);  
        _tprintf(_T("Key: %s\n"), webApp.szApiKey);  
        _tprintf(_T("LUID: %s\n"), webApp.szLuid);  
        _tprintf(_T("Tokens: %lu\n"), webApp.dwTokens);  
        _tprintf(_T("\n"));  
    }  
    while (pStorage->GetNextApplication(&webApp));  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFirstApplication](#), [RegisterAppId](#), [UnregisterAppId](#)

GetNextObject Method

```
BOOL GetNextObject(  
    LPWEB_STORAGE_OBJECT lpObject  
);
```

The **GetNextObject** method returns information about the next storage object that matches the specified label or content type.

Parameters

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain information about the storage object when the function returns. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetNextObject** method returns information about the next object that matches the label and/or content type that was specified by the **GetFirstObject** method. This method may only be called after **GetFirstObject** has been called, otherwise it will fail.

When all matching objects have been returned to the caller, this method will return zero (FALSE) and the **GetLastError** method will return NO_ERROR. Any other error code indicates an underlying problem with the request, such as an invalid parameter passed to the function.

You cannot intermix calls between **GetNextObject** and **EnumObjects**. The **EnumObjects** method will reset the internal object cache for the client session and subsequent calls to **GetNextObject** will fail.

Example

```
WEB_STORAGE_OBJECT webObject;  
  
if (pStorage->GetFirstObject(&webObject))  
{  
    do  
    {  
        // Print information about each object  
        _tprintf(_T("Object: %s\n"), webObject.szObjectId);  
        _tprintf(_T("Label: %s\n"), webObject.szLabel);  
        _tprintf(_T("Size: %lu\n"), webObject.dwObjectSize);  
        _tprintf(_T("Digest: %s\n"), webObject.szDigest);  
        _tprintf(_T("Content: %s\n"), webObject.szContent);  
        _tprintf(_T("\n"));  
    }  
    while (pStorage->GetNextObject(&webObject));  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[EnumObjects](#), [GetFirstObject](#)

CWebStorage::GetObjectInformation Method

```
BOOL GetObjectInformation(  
    LPCTSTR lpszObjectLabel,  
    LPWEB_STORAGE_OBJECT lpObject  
);
```

The **GetObjectInformation** method retrieves the metadata for a storage object.

Parameters

lpszObjectLabel

A pointer to a null terminated string that specifies the label of the storage object.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain additional information about the object. This parameter cannot be a NULL pointer.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetObjectInformation** method is used to retrieve the metadata for a stored object on the server, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents. This method can also be used as a simple method to determine if the specified object exists without the overhead of requesting the server attempt to retrieve the contents of an object.

Although storage object labels are similar to Windows file names, they are case-sensitive. When requesting information about an object, your application must specify the label name exactly as it was created. The object label cannot contain wildcard characters.

If you are only interested in obtaining the size of a stored object, you can use the **GetObjectSize** method.

To obtain information about how much storage your applications are using and the total number of stored objects, use the **GetStorageQuota** method.

Example

```
WEB_STORAGE_OBJECT webObject;  
  
// Get information about the object  
if (pStorage->GetObjectInformation(lpszObjectLabel, &webObject))  
{  
    // The object exists, display the metadata  
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);  
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);  
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);  
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);  
    _tprintf(_T("Content: %s\n"), webObject.szContent);  
}  
else  
{  
    // The object does not exist, display the error  
    CString strError;
```

```
    pStorage->GetLastError(strError);
    _tprintf(_T("Unable to get information on \"%s\" (%s)\n"), lpszObjectLabel,
(LPCTSTR)strError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFile](#), [GetData](#), [GetObjectSize](#), [GetStorageQuota](#), [PutFile](#), [PutData](#)

CWebStorage::GetObjectSize Method

```
DWORD GetObjectSize(  
    LPCTSTR lpszObjectLabel  
);  
  
BOOL GetObjectSize(  
    LPCTSTR lpszObjectLabel,  
    LPDWORD lpdwObjectSize  
);
```

The **GetObjectSize** method returns the size of the stored object.

Parameters

lpszObjectLabel

A pointer to a null terminated string that specifies the label of the storage object. This parameter cannot be a NULL pointer.

lpdwObjectSize

A pointer to an unsigned integer value that will contain the size of the object. If this parameter is NULL, the parameter is ignored and the method only checks for the existence of an object that matches the specified label.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **GetObjectSize** method is used to retrieve the size of a stored object on the server. This method can also be used as a simple method to determine if the specified object exists. An object size of zero means the object exists, but currently has no data associated with it.

If the first version of this method is called, where the object size is a return value, you can distinguish between an error and an empty object by calling the **GetLastError** method. If the last error code is NO_ERROR then the object exists, but it has no data associated with it. Any other error code value indicates the reason the object size could not be returned.

If the second version of this method is used and the *lpdwObjectSize* parameter is not NULL, its value is initialized to zero when the method is called, and updated with the object size when the method returns.

Although storage object labels are similar to Windows file names, they are case-sensitive. When requesting the size of an object, your application must specify the label name exactly as it was created. The object label cannot contain wildcard characters.

The **GetObjectInformation** method can be used to obtain the metadata associated with the storage object, including the size, content type, creation date and a SHA-256 digest of the data.

To obtain information about how much storage your applications are using and the total number of stored objects, use the **GetStorageQuota** method.

Example

```
// Check if the object exists  
DWORD dwObjectSize = 0;
```

```
if (pStorage->GetObjectSize(lpszObjectLabel, &dwObjectSize))
{
    // The object exists, display the size in bytes
    _tprintf(_T("The size of \"%s\" is %lu bytes\n"), lpszObjectLabel,
dwObjectSize);
}
else
{
    // The object size could not be determined, display the error
    CString strError;

    pStorage->GetErrorString(strError);
    _tprintf(_T("Unable to get the size of \"%s\" (%s)\n"), lpszObjectLabel,
(LPCTSTR)strError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetData](#), [GetObjectInformation](#), [GetStorageQuota](#), [PutData](#)

CWebStorage::GetStorageId Method

```
BOOL GetStorageId(  
    LPCTSTR lpszStorageId,  
    INT nMaxLength  
);  
  
BOOL GetStorageId(  
    CString & strStorageId  
);
```

The **GetStorageId** method returns the current storage container ID.

Parameters

lpszStorageId

A pointer to a null-terminated string that will contain the storage ID when the method returns. This parameter cannot be NULL and must be at least 35 characters in length. An alternate version of this method accepts a **CString** parameter if the application is compiled with support for MFC or ATL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the *lpszStorageId* string parameter, including the terminating null character.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The storage ID is a string that identifies the storage container that was opened with the **OpenStorage** method. The storage ID is associated with your account ID and development license and is guaranteed to be a unique value.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetAccountId](#), [OpenStorage](#)

CWebStorage::GetStorageQuota Method

```
BOOL GetStorageQuota(  
    LPWEB_STORAGE_QUOTA lpQuota  
);
```

The **GetStorageQuota** method returns quota limits assigned to your development account.

Parameters

lpQuota

A pointer to a [WEB_STORAGE_QUOTA](#) structure that will contain information the quota limits for your account when the method returns. This parameter cannot be NULL.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **GetStorageQuota** method can be used to determine how much storage space is available to your application. The *ulBytesFree* structure member will tell you how many bytes of storage you have available, and the *dwObjectSize* member will tell you the maximum size of any individual storage object. In addition to the bytes allocated for storage, there is also a limit on the total number of objects that your application may create which is specified by the *dwObjectLimit* member.

Accounts that are created with an evaluation license have much lower quota limits than a standard account and should be used for testing purposes only. After the evaluation period has ended, all objects stored using the evaluation license will be deleted.

Example

```
CWebStorage appStorage;  
WEB_STORAGE_QUOTA appQuota;  
  
// Display storage account usage and limits  
if (appStorage.GetStorageQuota(&appQuota))  
{  
    _tprintf(_T("Objects Used: %lu\n"), appQuota.dwObjects);  
    _tprintf(_T("Object Limit: %lu\n"), appQuota.dwObjectLimit);  
    _tprintf(_T("Object Size: %lu\n"), appQuota.dwObjectSize);  
    _tprintf(_T("Bytes Used: %I64u\n"), appQuota.ulBytesUsed);  
    _tprintf(_T("Bytes Free: %I64u\n"), appQuota.ulBytesFree);  
    _tprintf(_T("Storage Limit: %I64u\n"), appQuota.ulStorageLimit);  
}  
else  
{  
    // Unable to get the quota information for this account  
    CString strError;  
  
    appStorage.GetErrorString(strError);  
    _tprintf(_T("Unable to get the account quota (%s)\n"), (LPCTSTR)strError);  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetObjectInformation](#), [GetObjectSize](#), [WEB_STORAGE_QUOTA](#)

CWebStorage::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the storage server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The default timeout period is 10 seconds.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

See Also

[OpenStorage](#), [SetTimeout](#)

CWebStorage::GetTransferStatus Method

```
LPWEB_STORAGE_TRANSFER lpStatus  
);
```

The **GetTransferStatus** method returns information about the current data transfer in progress.

Parameters

lpStatus

A pointer to a [WEB_STORAGE_TRANSFER](#) structure which contains information about the status of the current data transfer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **GetTransferStatus** method returns information about the current data transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no data currently being transferred, this method will return the status of the last successful data transfer.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the storage session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Cancel](#), [GetData](#), [PutData](#), [RegisterEvent](#)

CWebStorage::IsBlocking Method

BOOL IsBlocking();

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can potentially allow the application to be re-entered, it is possible that another blocking method may be called while it is in progress. Because only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the storage client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

If your application attempts to perform multiple operations simultaneously, such as downloading multiple stored objects to the local system, each client session should be isolated in its own thread. This will avoid potential situations where a request is refused because a blocking data transfer is already in progress on the current thread.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Cancel](#), [GetFile](#), [GetData](#), [IsConnected](#), [IsInitialized](#), [PutFile](#), [PutData](#)

CWebStorage::IsConnected Method

BOOL IsConnected();

The **IsConnected** method is used to determine if a connection has been established with the storage server.

Parameters

None.

Return Value

If the client is connected to a server, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero. To get extended error information, call the **GetLastError** method.

Remarks

The client does not maintain a continuous, persistent connection with the storage server. The connection may be closed and reopened internally as needed. If the client session has been idle for a period of time, this method can return zero.

If the **GetLastError** method returns `ST_ERROR_NOT_CONNECTED` it means the client session is valid, however it not currently connected to the storage server. The next call to store or retrieve an object will the cause the client to reconnect automatically.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cswebv10.lib`

See Also

[IsBlocking](#), [IsInitialized](#)

CWebStorage::IsInitialized Method

BOOL IsInitialized();

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on a different system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

See Also

[CWebStorage](#), [IsBlocking](#), [IsConnected](#)

CWebStorage::MoveObject Method

```
BOOL MoveObject(  
    LPCTSTR lpszOldLabel,  
    LPCTSTR lpszNewLabel,  
    DWORD dwStorageType,  
    LPWEB_STORAGE_OBJECT lpObject  
);
```

The **MoveObject** method moves an existing object to a different storage container.

Parameters

lpszOldLabel

A pointer to a null terminated string which specifies the name of the existing storage object to be moved. This parameter must specify a valid object label and cannot be a NULL pointer or an empty string.

lpszNewLabel

A pointer to a null terminated string which specifies a new label for the storage object being moved. This parameter may be NULL or point to an empty string, in which case the label name is not changed.

dwStorageType

An integer value that identifies the storage container type. If this parameter is omitted, the object is renamed and not moved to another container. One of the following values may be specified:

Constant	Description
WEB_STORAGE_GLOBAL (1)	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN (2)	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
WEB_STORAGE_MACHINE (3)	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER (4)	Current user storage. Objects stored using this storage type are only available to the current user logged in on

the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain information about the new storage object. If this information is not required, this parameter may be omitted or a NULL pointer.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **MoveObject** method is used to move an existing storage object to a new container. For example, it can move an object originally created using `WEB_STORAGE_USER` to the `WEB_STORAGE_MACHINE` container. The *dwStorageType* parameter must specify a valid storage container type and cannot be `WEB_STORAGE_DEFAULT`.

If the *dwStorageType* parameter specifies the same container that the object is currently in, and the *lpzNewLabel* parameter specifies a new label name, this method will simply rename the existing object and is effectively the same as calling the **RenameObject** method.

To duplicate an existing storage object, use the **CopyObject** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cswebv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CopyObject](#), [DeleteObject](#), [RenameObject](#)

CWebStorage::OpenStorage Method

```
BOOL OpenStorage(  
    DWORD dwStorageType  
);
```

```
BOOL OpenStorage(  
    LPCTSTR lpszAppId,  
    DWORD dwStorageType  
);
```

The **OpenStorage** method establishes a connection with the server and opens the storage container associated with the specified storage type.

Parameters

lpszAppId

A pointer to null terminated string which specifies the application ID for the storage container. The application ID is a string that uniquely identifies the application and can only contain letters, numbers, the period and the underscore character. If this parameter is omitted, NULL or an empty string, the default identifier **SocketTools.Storage.Default** will be used.

dwStorageType

An integer value that identifies the storage container type. If this parameter is omitted, the WEB_STORAGE_GLOBAL container will be opened. One of the following values may be specified:

Constant	Description
WEB_STORAGE_GLOBAL (1)	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN (2)	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
WEB_STORAGE_MACHINE (3)	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER (4)	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the

Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

Return Value

If the method succeeds, the return value is a handle to the storage container. If the method fails, the return value is `INVALID_HANDLE`. To get extended error information, call the **GetLastError** method.

Remarks

The **OpenStorage** method opens the specified storage container and requests an access token for the application. This is the first method that must be called prior to accessing any stored objects.

The application ID is a string that uniquely identifies the application requesting the access and must have been previously registered with the server by calling the **RegisterAppId** method. If the *lpzAppId* parameter is `NULL` or an empty string, the method will use a default internal ID that is allocated for each storage account. You can use this default ID if you wish to share data between all of the applications you create.

The storage type specifies the type of container that objects will be stored in. In most cases, we recommend using `WEB_STORAGE_GLOBAL` which means that stored objects will be accessible to all users of your application. However, you can limit access to the stored objects based on the local domain, local machine ID or the current user SID.

If you specify anything other than global storage, objects can be orphaned if the system configuration changes. For example, if `WEB_STORAGE_MACHINE` is specified the objects that are stored can only be accessed from that system. If the system is reconfigured (for example, the boot volume formatted and Windows is reinstalled) the unique identifier for that system will change and the previous objects that were stored by your application can no longer be accessed.

It is advisable to store critical application data and configuration information using `WEB_STORAGE_GLOBAL` and use other non-global storage containers for configuration information that is unique to that system and/or user which is not critical and can be easily recreated.

If a storage container was previously opened, calling this method will automatically close the current container and open the new container specified by the *dwStorageType* parameter. This can change the internal handle value used to reference the container and the old handle will no longer be valid.

Example

```
CWebStorage appStorage;
LPCTSTR lpzAppId = _T("MyCompany.WebTest.1");
LPCTSTR lpzLocalFile = _T("TestDocument.pdf");
LPCTSTR lpzObjectLabel = _T("TestDocument.pdf");
WEB_STORAGE_OBJECT webObject = { 0, };

// Register the application ID which identifies the application
if (!appStorage.RegisterAppId(lpzAppId))
{
    tprintf(_T("Unable to register the application ID"));
    _exit(0);
}
```

```

// Open the application storage container
if (!appStorage.OpenStorage(lpszAppId, WEB_STORAGE_GLOBAL);
{
    _tprintf(_T("Unable to open global storage for this application"));
    _exit(0);
}

// Upload a local file to the storage server and return information
// about the stored object when it completes
if (appStorage.PutFile(lpszLocalFile, lpszObjectLabel, &webObject))
{
    // Print information about the object that was created
    _tprintf(_T("Object: %s\n"), webObject.szObjectId);
    _tprintf(_T("Label: %s\n"), webObject.szLabel);
    _tprintf(_T("Size: %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest: %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The upload failed, display error information
    CString strError;

    appStorage.GetLastError(strError);
    _tprintf(_T("Cannot store \"%s\" (%s)\n"), lpszLocalFile,
(LPCTSTR)strError);
}

// Release the handle allocated for this storage session
appStorage.CloseStorage();

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CWebStorage::CloseStorage](#), [CWebStorage::GetData](#), [CWebStorage::GetFile](#),
[CWebStorage::GetStorageId](#), [CWebStorage::PutData](#), [CWebStorage::PutFile](#),
[CWebStorage::RegisterAppId](#)

CWebStorage::PutData Method

```
BOOL PutData(  
    LPCTSTR lpszObjectLabel,  
    LPCVOID lpvBuffer,  
    DWORD dwLength,  
    LPWEB_STORAGE_OBJECT LpObject  
);
```

```
BOOL PutData(  
    LPCTSTR lpszObjectLabel,  
    LPCTSTR lpszObjectText,  
    INT cchObjectText,  
    LPWEB_STORAGE_OBJECT LpObject  
);
```

```
BOOL PutData(  
    LPCTSTR lpszObjectLabel,  
    LPCVOID lpvBuffer,  
    DWORD dwLength,  
    LPCTSTR lpszContentType,  
    DWORD dwAttributes,  
    LPWEB_STORAGE_OBJECT LpObject  
);
```

The **PutData** method stores the contents of buffer to a container and returns information about the new object.

Parameters

lpszObjectLabel

A pointer to a null terminated string that specifies the label of the storage object that will be created or replaced. This parameter cannot be NULL or a zero-length string. The method will fail if the label contains any illegal characters.

lpvBuffer

A pointer to a buffer that contains the data to be stored. If this parameter is NULL, the *dwLength* parameter must have a value of zero and a storage object will be created which has no data associated with it.

dwLength

An unsigned integer value that specifies the number of bytes that will be copied from the *lpvBuffer* parameter and stored in the object. If the *lpvBuffer* parameter is NULL, this value must be zero or the method will fail.

lpszObjectText

A pointer to a null terminated string which is used with an alternate version of this method. The string will be stored as a text object and the *cchObjectText* parameter will specify the number of characters to be stored.

cchObjectText

An integer value that specifies the number of characters that will be copied from the *lpszObjectText* string and stored in the object. If the *lpszObjectText* parameter is NULL, this value must be zero or the method will fail. If this parameter is omitted, the length of the string will be calculated by counting the number of characters up to the terminating null.

lpzContentType

A pointer to a null terminated string that identifies the contents of the buffer being stored. If this parameter is omitted, a NULL pointer, or specifies a zero-length string, the method will attempt to automatically determine the content type based on the object label and the contents of the buffer.

dwAttributes

An unsigned integer that specifies the attributes associated with the storage object. If this parameter is omitted, the default value `WEB_OBJECT_NORMAL` will be used. This value can be a combination of one or more of the following bitflags using a bitwise OR operation:

Value	Constant	Description
0	<code>WEB_OBJECT_DEFAULT</code>	Default object attributes. This value is used to indicate the object can be modified, or that the attributes for a previously existing object should not be changed.
1	<code>WEB_OBJECT_NORMAL</code>	A normal object that that can be read and modified by the application. This is the default attribute for new objects that are created by the application.
2	<code>WEB_OBJECT_READONLY</code>	A read-only object that can only be read by the application. Attempts to modify or replace the contents of the object will fail. Read-only objects can be deleted.
4	<code>WEB_OBJECT_HIDDEN</code>	A hidden object. Objects with this attribute are not returned when enumerated using the EnumObjects method. The object can only be accessed directly when specifying its label.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain additional information about the object that was created or replaced. This parameter may be omitted or NULL if the information is not required.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **PutData** method uploads the contents of a buffer to the current storage container. The content type, which identifies the type of data stored in the object, and its attributes may be specified by the caller or default values may be used.

If a content type is provided, it must specify a valid MIME media type and subtype. For example, normal text files have a content type of **text/plain** while an XML-formatted text file would have a content type of **text/xml**. Files that contain unstructured binary data are typically identified as **application/octet-stream**.

If the label identifies an object that already exists in the container, and that object was created with the `WEB_OBJECT_READONLY` attribute, this method will fail. To replace a read-only object, the application must explicitly move, rename or delete the existing object.

If **PutData** method is used to store the contents of a string as a text object, the string should be terminated with a null character and cannot contain embedded nulls. If the Unicode version of this function is called, the contents of the string will be normalized prior to being converted to UTF-8 using canonical composition, where decomposed characters are combined to create their canonical precomposed equivalent.

The size of the text object that is created can be different from the length of the string buffer passed to this method, depending on how the text has been encoded and stored. This will almost always be the case when the Unicode version of this function is called because the UTF-16 string will be converted and stored as UTF-8 encoded text. This encoding will typically increase the size of the stored object unless the string only contains ASCII characters.

The version of the **PutData** method which accepts a string should only be used to store textual data in a null terminated string, and should never be used to store binary data. If you wish to create or update an object which contains binary data, you should always use use a byte array.

If you want to upload the contents of a file, the **PutFile** method simplifies this process. The example code below demonstrates how the Windows API can be used to read from a local file and store the contents using **PutData**.

If you are storing a large amount of data and want your application to receive progress updates during the data transfer, use the **RegisterEvent** method and provide a pointer to a static callback method that will receive event notifications.

Example

```
HANDLE hFile = INVALID_HANDLE_VALUE;
LPBYTE lpContents = NULL;
DWORD dwLength = 0;
BOOL bFileRead = FALSE;
WEB_STORAGE_OBJECT webObject;

// Open a file on the local system and read the contents
// into a buffer that will be stored on the server
hFile = CreateFile(lpszLocalFile,
                  GENERIC_READ,
                  0,
                  NULL,
                  OPEN_EXISTING,
                  FILE_ATTRIBUTE_NORMAL,
                  NULL);

if (hFile == INVALID_HANDLE_VALUE)
{
    // Unable to open the file
    return;
}

// Get the size of the file and allocate a buffer large
// enough to store the contents of the file
dwLength = GetFileSize(hFile, NULL);
lpContents = (LPBYTE)LocalAlloc(LPTR, dwLength + 1);

if (lpContents == NULL)
{
    // Memory allocation failed
    return;
}
```

```

bFileRead = ReadFile(hFile, lpContents, dwLength, &dwLength, NULL);
CloseHandle(hFile);

if (!bFileRead)
{
    // Unable to read the contents of the file
    return;
}

// Store the contents of the buffer, identifying it as an unstructured
// stream of bytes, and the object is created with read/write access
if (pStorage->PutData(lpszObjectLabel, lpContents, dwLength, &webObject))
{
    // The object was created or replaced, display the metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The object could not be created
    CString strError;

    pStorage->GetLastError(strError);
    _tprintf(_T("Unable to create \"%s\" (%s)\n"), lpszObjectLabel,
(LPCTSTR)strError);
}

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetFile](#), [GetData](#), [PutFile](#), [RegisterEvent](#)

CWebStorage::PutFile Method

```
BOOL PutFile(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszObjectLabel,  
    LPWEB_STORAGE_OBJECT lpObject  
  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszObjectLabel,  
    LPCTSTR lpszContentType,  
    DWORD dwAttributes,  
    LPWEB_STORAGE_OBJECT lpObject  
);
```

The **PutFile** method uploads the contents of a local file to a storage container and returns information about the new object.

Parameters

lpszLocalFile

A pointer to a null terminated string that specifies the name of the local file that will be uploaded. If a path is not specified, the file will be read from the current working directory. The current user must have read access to the file, and an error will be returned if the method cannot obtain an exclusive lock on the file during the upload process.

lpszObjectLabel

A pointer to a null terminated string that specifies the label of the storage object that will be created or replaced. This parameter cannot be NULL or a zero-length string. The method will fail if the label contains any illegal characters.

lpszContentType

A pointer to a null terminated string that identifies the contents of the file being uploaded. If this parameter is omitted, a NULL pointer, or specifies a zero-length string, the method will attempt to automatically determine the content type based on the file name extension and the contents of the file.

dwAttributes

An unsigned integer that specifies the attributes associated with the storage object. If this parameter is omitted, a default value of WEB_OBJECT_NORMAL will be used. This value can be a combination of one or more of the following bitflags using a bitwise OR operation:

Value	Constant	Description
0	WEB_OBJECT_DEFAULT	Default object attributes. This value is used to indicate the object can be modified, or that the attributes for a previously existing object should not be changed.
1	WEB_OBJECT_NORMAL	A normal object that that can be read and modified by the application. This is the default attribute for new objects that are created by the application.
2	WEB_OBJECT_READONLY	A read-only object that can only be read by the application. Attempts to modify or replace the

		contents of the object will fail. Read-only objects can be deleted.
4	WEB_OBJECT_HIDDEN	A hidden object. Objects with this attribute are not returned when enumerated using the EnumObjects method. The object can only be accessed directly when specifying its label.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain additional information about the object. This parameter may be omitted or NULL if the information is not required.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **PutFile** method uploads the contents of a local file to the current storage container. The content type, which identifies the type of data stored in the object, and its attributes may be specified by the caller or default values may be used.

If a content type is provided, it must specify a valid MIME media type and subtype. For example, normal text files have a content type of **text/plain** while an XML-formatted text file would have a content type of **text/xml**. Files that contain unstructured binary data are typically identified as **application/octet-stream**. If the content type is not explicitly specified, an attempt will be made to identify it automatically based on contents of the file and the file extension.

If the label identifies an object that already exists in the container, and that object was created with the WEB_OBJECT_READONLY attribute, this method will fail. To replace a read-only object, the application must first move, rename or delete the existing object.

The **ValidateLabel** method can be used to ensure a label is valid prior to calling this method. Refer to that method for more information about the difference between Windows file names and object labels

If you are uploading a large file and want your application to receive progress updates during the data transfer, use the **RegisterEvent** method and provide a pointer to a static callback method that will receive event notifications.

Example

```

WEB_STORAGE_OBJECT webObject;

// Upload a local file to the storage container, automatically
// determining the content type with normal read/write access
if (pStorage->PutFile(lpszLocalFile, lpszObjectLabel, &webObject))
{
    // The file was uploaded, display the object metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The file could not be uploaded, display the error

```

```
    CString strError;

    pStorage->GetLastError(strError);
    _tprintf(_T("Unable to store \"%s\" (%s)\n"), lpszLocalFile,
(LPCTSTR)strError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DownloadFile](#), [GetFile](#), [RegisterEvent](#), [UploadFile](#), [ValidateLabel](#)

CWebStorage::RegisterAppId Method

```
BOOL RegisterAppId(  
    LPCTSTR lpszAppId  
);
```

The **RegisterAppId** method registers a unique application identifier with the server.

Parameters

lpszAppId

A pointer to a null terminated string which identifies the application requesting access. This parameter cannot be NULL or point to a zero-length string. If the application ID contains illegal characters, the method will fail. See the remarks below on the recommended method for identifying your application.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **RegisterAppId** method registers an application ID with the server which uniquely identifies the application that is requesting access to the storage container. The ID must only consist of ASCII letters, numbers, the period and underscore character. Whitespace characters and non-ASCII Unicode characters are not permitted. The maximum length of an application ID string is 64 characters, including the terminating null character.

It is recommended that you use a standard format for the application ID that consists of your company name, application name and optionally a version number. For example:

- MyCompany.MyApplication
- MyCompany.MyApplication.1

It is important to note that with these two example IDs, although they are similar, they reference two different applications. Objects stored using the first ID will not be accessible using the second ID. If you want to store objects that should be shared between all versions of the application, it is recommended that you use the first form, without the version number. If you want to store objects that should only be accessible to a specific version of your application, then it is recommended that you use the second form that includes the version number.

It is safe to call this method with an application ID that was previously registered. If the provided application ID has already been registered, this method will succeed. You can choose to call this method every time your application starts to ensure the application has been registered correctly.

If you no longer wish to use an application ID you have previously registered, you can call the **UnregisterAppId** method. Exercise caution when unregistering an application. This will cause all objects stored using that ID to be deleted by the storage server. Once an application ID has been unregistered, the operation is permanent. Calling **UnregisterAppId** and then **RegisterAppId** again using the same ID will force the system to create new access tokens for your application. You will not be able to regain access to the objects that were previously stored using that ID.

The application ID is intended to be an application defined human-readable string that uniquely identifies your application. If you want to obtain the internal storage ID associated with your application, use the **GetStorageId** method. The storage ID is a fixed-length string of letters and

numbers guaranteed to be unique across all applications that you register.

It is not required for your application to create a unique application ID. Each storage account has a default internal application ID named **SocketTools.Storage.Default**. This default ID is used if a NULL pointer or an empty string is specified to methods like **OpenStorage**. It is intended to identify storage available to all applications that you create.

To enumerate the application IDs registered with your storage account, use the **GetFirstApplication** and **GetNextApplication** methods.

Example

```
CWebStorage appStorage;
LPCTSTR lpszAppId = _T("MyCompany.WebTest.1");
LPCTSTR lpszLocalFile = _T("TestDocument.pdf");
LPCTSTR lpszObjectLabel = _T("TestDocument.pdf");
WEB_STORAGE_OBJECT webObject = { 0, };

// Register the application ID which identifies the application
if (!appStorage.RegisterAppId(lpszAppId))
{
    tprintf(_T("Unable to register the application ID"));
    _exit(0);
}

// Open the application storage container
if (!appStorage.OpenStorage(lpszAppId, WEB_STORAGE_GLOBAL);
{
    tprintf(_T("Unable to open global storage for this application"));
    _exit(0);
}

// Upload a local file to the storage server and return information
// about the stored object when it completes
if (appStorage.PutFile(lpszLocalFile, lpszObjectLabel, &webObject))
{
    // Print information about the object that was created
    _tprintf(_T("Object: %s\n"), webObject.szObjectId);
    _tprintf(_T("Label: %s\n"), webObject.szLabel);
    _tprintf(_T("Size: %I64u\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest: %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The upload failed, display error information
    CString strError;

    appStorage.GetLastError(strError);
    _tprintf(_T("Unable to upload \"%s\" (%s)\n"), lpszLocalFile,
(LPCTSTR)strError);
}

// Release the handle allocated for this storage session
appStorage.CloseStorage();
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CloseStorage](#), [GetFirstApplication](#), [GetNextApplication](#), [GetStorageId](#), [OpenStorage](#),
[UnregisterAppId](#), [ValidateAppId](#)

CWebStorage::RegisterEvent Method

```
BOOL RegisterEvent(  
    UINT nEventId,  
    WEBEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback method. This parameter cannot be zero. The following values may be used:

Constant	Description
WEB_EVENT_CONNECT (1)	The connection to the storage server has been established and the session has been authenticated. This is the first event that occurs when initiating an operation to create or retrieve a storage object.
WEB_EVENT_DISCONNECT (2)	The connection to the storage server has been closed and the session is terminating. This is the last event that occurs after completing an operation to create or retrieve a storage object.
WEB_EVENT_READ (4)	The contents of an object is being read from the storage container. This event occurs only once after the operation has been initiated by the application. This event can also be generated if there is an error during the process of retrieving the object contents from the container.
WEB_EVENT_WRITE (8)	The contents of an object is being written to the storage container. This event occurs only once after the operation has been initiated by the application. This event can also be generated if there is an error during the process of submitting the object contents to the container.
WEB_EVENT_TIMEOUT (16)	The operation has exceeded the specified timeout period. The application may attempt to retry the operation or report an error to the user. This event typically indicates a connectivity problem with the storage server.
WEB_EVENT_CANCEL (32)	The operation has been canceled. This event occurs after the application calls the Cancel method while an object is being stored or retrieved.
WEB_EVENT_PROGRESS (64)	A storage operation is in progress. This event periodically occurs as the contents of a storage object is being read or written from the container. To retrieve information about the status of the operation, the application should register a handler for this event and call the GetTransferStatus method from within that handler.

lpEventProc

Specifies the procedure-instance address of the application defined callback method. For more information about the callback method, see the description of the **WebEventProc** callback method. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback method specified by *lpEventProc*. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **RegisterEvent** method associates a static callback method with a specific event. The event handler is a **WebEventProc** function that is invoked when the event occurs. Arguments are passed to the method to identify the storage handle, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

Events are only generated as the result of a call to the **GetFile**, **GetData**, **PutFile** and **PutData** methods. Other storage methods will not generate event notifications.

This method is typically used to register an event handler that is invoked while the contents of a storage object is being transferred. The WEB_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one WEB_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a DWORD_PTR value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Cancel](#), [GetTransferStatus](#), [UnregisterEvent](#), [WebEventProc](#)

CWebStorage::RenameObject Method

```
BOOL RenameObject(  
    LPCTSTR lpszOldLabel,  
    LPCTSTR lpszNewLabel  
);
```

The **RenameObject** method changes the label associated with the storage object.

Parameters

lpszOldLabel

A pointer to a null terminated string which specifies the name of the existing storage object to be renamed. This parameter must specify a valid object label and cannot be a NULL pointer or an empty string.

lpszNewLabel

A pointer to a null terminated string which specifies a new label for the storage object being moved. This parameter may not be NULL or point to an empty string.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **RenameObject** method is used to change the label for an existing storage object. Although storage object labels are similar to Windows file names, they are case-sensitive. When renaming an object, your application must specify the original label name exactly as it was created. The object label cannot contain wildcard characters.

To duplicate an existing storage object, use the **CopyObject** method. If you need to move the object to a different storage container, use the **MoveObject** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CopyObject](#), [DeleteObject](#), [MoveObject](#)

CWebStorage::ResetStorage Method

```
BOOL ResetStorage();
```

The **ResetStorage** method resets the application storage container and deletes all stored objects.

Parameters

None.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The storage container contains information for each of the objects that have been stored using the handle returned by the **OpenStorage** method. Each of these objects are associated with both the application ID and the storage type that was specified by the caller. This method instructs the server to reset the container back to its initial state, deleting all of the objects that were stored in it.



Exercise caution when using this method. The reset operation is immediate and the objects that are stored in the container are permanently deleted. They cannot be recovered by your application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DeleteObject](#), [OpenStorage](#), [RegisterAppId](#), [UnregisterAppId](#)

CWebStorage::SetLastError Method

```
VOID SetLastError(  
    DWORD dwErrorCode  
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Method failure is typically indicated by a return value such as FALSE, NULL, INVALID_HANDLE or WEBAPI_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. An application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cswebv10.lib

See Also

[GetErrorString](#), [GetLastError](#), [ShowError](#)

CWebStorage::SetTimeout Method

```
DWORD SetStorageTimeout(  
    HSTORAGE hStorage,  
    DWORD dwTimeout  
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the storage server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

hClient

Handle to the client session.

dwTimeout

The number of seconds to wait for a storage operation to complete.

Return Value

If the method succeeds, the return value is the previous timeout period for the session. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The default timeout period is 10 seconds. The minimum timeout period is 5 seconds, and the maximum timeout period is 60 seconds. Values outside of this range will be normalized internally and will not cause the method to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

See Also

[GetTimeout](#), [OpenStorage](#)

CWebStorage::ShowError Method

```
INT ShowError(  
    LPCTSTR lpszAppTitle,  
    UINT uType,  
    DWORD dwErrorCode  
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A pointer to a string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** method. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetErrorString](#), [GetLastError](#), [SetLastError](#)

CWebStorage::UnregisterAppId Method

```
BOOL UnregisterAppId(  
    LPCTSTR lpszAppId  
);
```

The **UnregisterAppId** method unregisters the application identifier and deletes all associated storage objects.

Parameters

lpszAppId

A pointer to a null terminated string which specifies the application ID to be deleted. This parameter cannot be NULL or point to a zero-length string. If the application ID contains illegal characters, the method will fail.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **UnregisterAppId** method deletes the internal storage identifier associated with the application ID and revokes all access tokens that were granted for the application. This operation is immediate and permanent.



Exercise caution when using this method. This will permanently delete all objects that were stored for the specified application. Calling **UnregisterAppId** and then **RegisterAppId** again using the same ID will force the system to create new access tokens for your application. You will not be able to regain access to the objects that were previously stored using that ID.

This method cannot be used to unregister the default storage application ID **SocketTools.Storage.Default**. If this ID is specified, the method will fail with an error indicating that the ID is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cswebv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[OpenStorage](#), [RegisterAppId](#), [ResetStorage](#), [ValidateAppId](#)

CWebStorage::UnregisterEvent Method

```
BOOL RegisterEvent(  
    UINT nEventId  
);
```

The **UnregisterEvent** method removes an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback method. This parameter cannot be zero. The following values may be used:

Constant	Description
WEB_EVENT_CONNECT (1)	The connection to the storage server has been established and the session has been authenticated. This is the first event that occurs when initiating an operation to create or retrieve a storage object.
WEB_EVENT_DISCONNECT (2)	The connection to the storage server has been closed and the session is terminating. This is the last event that occurs after completing an operation to create or retrieve a storage object.
WEB_EVENT_READ (4)	The contents of an object is being read from the storage container. This event occurs only once after the operation has been initiated by the application. This event can also be generated if there is an error during the process of retrieving the object contents from the container.
WEB_EVENT_WRITE (8)	The contents of an object is being written to the storage container. This event occurs only once after the operation has been initiated by the application. This event can also be generated if there is an error during the process of submitting the object contents to the container.
WEB_EVENT_TIMEOUT (16)	The operation has exceeded the specified timeout period. The application may attempt to retry the operation or report an error to the user. This event typically indicates a connectivity problem with the storage server.
WEB_EVENT_CANCEL (32)	The operation has been canceled. This event occurs after the application calls the Cancel method while an object is being stored or retrieved.
WEB_EVENT_PROGRESS (64)	A storage operation is in progress. This event periodically occurs as the contents of a storage object is being read or written from the container. To retrieve information about the status of the operation, the application should register a handler for this event and call the GetTransferStatus method from within that handler.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **UnregisterEvent** method removes the association between a specific event and its callback function. This method can be used if the application no longer requires the event notification. All event handlers for the client session are automatically removed when the **CloseStorage** method is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Cancel](#), [GetTransferStatus](#), [RegisterEvent](#), [WebEventProc](#),

CWebStorage::UploadFile Method

```
BOOL UploadFile(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszObjectLabel,  
    LPWEB_STORAGE_OBJECT LpObject  
);  
  
BOOL UploadFile(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszObjectLabel,  
    LPCTSTR lpszAppId,  
    DWORD dwStorageType,  
    DWORD dwAttributes,  
    DWORD dwTimeout,  
    LPWEB_STORAGE_OBJECT LpObject,  
    WEBEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

The **UploadFile** method uploads the contents of a local file and creates or overwrites a storage object.

Parameters

lpszLocalFile

A pointer to a null terminated string that specifies the name of the local file that will be created or overwritten with the contents of the storage object. If a path is not specified, the file will be created in the current working directory.

lpszObjectLabel

A pointer to a null terminated string that specifies the label of the object that should be retrieved from the server.

lpszAppId

A pointer to null terminated string which specifies the application ID for the storage container. The application ID is a string that uniquely identifies the application and can only contain letters, numbers, the period and the underscore character. If this parameter is omitted, NULL or an empty string, the default identifier **SocketTools.Storage.Default** will be used.

dwStorageType

An integer value that identifies the storage container type. If this parameter is omitted, the default value of WEB_STORAGE_GLOBAL will be used. One of the following values may be specified:

Constant	Description
WEB_STORAGE_GLOBAL (1)	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN	Local domain storage. Objects stored using this storage

(2)	type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
WEB_STORAGE_MACHINE (3)	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER (4)	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

dwAttributes

An unsigned integer that specifies the attributes associated with the storage object. If this parameter is omitted, a default value of `WEB_OBJECT_NORMAL` will be used. This value can be a combination of one or more of the following bitflags using a bitwise OR operation:

Value	Constant	Description
0	<code>WEB_OBJECT_DEFAULT</code>	Default object attributes. This value is used to indicate the object can be modified, or that the attributes for a previously existing object should not be changed.
1	<code>WEB_OBJECT_NORMAL</code>	A normal object that that can be read and modified by the application. This is the default attribute for new objects that are created by the application.
2	<code>WEB_OBJECT_READONLY</code>	A read-only object that can only be read by the application. Attempts to modify or replace the contents of the object will fail. Read-only objects can be deleted.
4	<code>WEB_OBJECT_HIDDEN</code>	A hidden object. Objects with this attribute are not returned when enumerated using the <code>CWebStorage::EnumObjects</code> method. The object can only be accessed directly when specifying its label.

dwTimeout

An unsigned integer value that specifies a timeout period in seconds. If this value is zero, a default timeout period will be used.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain additional information about the object that has been downloaded. This parameter may be NULL if the information is not required.

lpEventProc

Specifies the procedure-instance address of the application defined callback method. For more information about the callback method, see the description of the **WebEventProc** callback function. If this parameter is NULL, no callback function will be invoked during the data transfer.

dwParam

A user-defined integer value that is passed to the callback function specified by *lpEventProc*. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **UploadFile** method uploads the contents of a local file and either creates a new storage object, or replaces an object if one already exists with the same label. The **UploadFile** method provides a simpler interface that defaults to uploading an object to the global storage container.

If the label identifies an object that already exists in the container, and that object was created with the WEB_OBJECT_READONLY attribute, this method will fail. To replace a read-only object, the application must explicitly move, rename or delete the existing object.

Additional metadata about the object will be returned in the WEB_STORAGE_OBJECT structure provided by the caller, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents.

If you are uploading a large file and want your application to receive progress updates during the data transfer, provide a pointer to a static callback function as the *lpEventProc* parameter. That function will receive event notifications as the data is being uploaded.

Example

```
CWebStorage appStorage;
WEB_STORAGE_OBJECT webObject;

// Upload a local file to the global storage container
if (appStorage.UploadFile(lpszLocalFile, lpszObjectLabel, &webObject))
{
    // The object was uploaded, display the metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The object could not be uploaded, display the error
    CString strError;

    appStorage.GetLastError(strError);
    _tprintf(_T("Unable to upload \"%s\" (%s)\n"), lpszLocalFile,
(LPCTSTR)strError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DownloadFile](#), [GetFile](#), [PutFile](#)

CWebStorage::ValidateAppId Method

```
BOOL ValidateAppId(  
    LPCTSTR lpszAppId  
);
```

The **ValidateAppId** method validates the specified application identifier.

Parameters

lpszAppId

A pointer to a null terminated string which specifies the application ID to be validated. This parameter cannot be NULL or point to a zero-length string. If the application ID contains illegal characters, the method will fail.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **ValidateAppId** method is used to determine if the specified application identifier is valid and has been previously registered using the **RegisterAppId** method. The ID must only consist of ASCII letters, numbers, the period and underscore character. Whitespace characters and non-ASCII Unicode characters are not permitted. The maximum length of an application ID string is 64 characters, including the terminating null character.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RegisterAppId](#), [UnregisterAppId](#), [ValidateLabel](#)

CWebStorage::ValidateLabel Method

```
BOOL ValidateLabel(  
    LPCTSTR lpszObjectLabel  
);
```

The **ValidateLabel** method validates the specified object label.

Parameters

lpszObjectLabel

A pointer to a null terminated string which specifies the object label to be validated. This parameter cannot be NULL or point to a zero-length string.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

Object labels are similar to Windows file names, except they are case-sensitive. The maximum length of a label string is 512 characters, including the terminating null character. Leading and trailing whitespace (spaces, tabs, linebreaks, etc.) are ignored in label names.

Illegal characters include ASCII and Unicode control characters 1 through 31, single quotes (39), double quotes (34), less than symbol (60), greater than symbol (62), pipe (124), asterisk (42) and question mark (63). A null character (0) specifies the end of the label and any subsequent characters are ignored. It is not possible to embed null characters in the label name.

Label names may contain forward slash (47) characters and backslash (92) characters, however it is important to note that objects are not stored in a hierarchical structure. An application can create its own folder-like structure to the labels it creates, but this structure is not imposed or enforced by the library.

If the application is built to use Unicode, labels can contain Unicode characters which are internally encoded as UTF-8. This is important to consider if you have an project built using a multi-byte (ANSI) character set and it needs to access an object that was created using Unicode characters. In that case, the ANSI application must be prepared to handle UTF-8 encoded names and display them appropriately.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CWebStorage::RegisterAppId](#), [CWebStorage::UnregisterAppId](#)

CWebStorage Data Structures

- SYSTEMTIME
- WEB_STORAGE_APPLICATION
- WEB_STORAGE_QUOTA
- WEB_STORAGE_OBJECT
- WEB_STORAGE_TRANSFER

SYSTEMTIME Structure

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include windows.h.

WEB_STORAGE_APPLICATION Structure

The **WEB_STORAGE_APPLICATION** structure contains information about a registered application.

```
typedef struct _WEB_STORAGE_APPLICATION
{
    TCHAR        szAppId[64];
    TCHAR        szApiKey[64];
    TCHAR        szLuid[64];
    DWORD        dwTokens;
    DWORD        dwReserved;
    SYSTEMTIME   stCreated;
    SYSTEMTIME   stUpdated;
} WEB_STORAGE_APPLICATION, *LPWEB_STORAGE_APPLICATION;
```

Members

szAppId

A null-terminated string which contains the application ID that was registered using the **RegisterAppId** method.

szApiKey

A null-terminated string which contains the API key value that is used internally by the storage services. The AppID is effectively the human readable alias for this key value used to identify the stored objects for the application. This value is guaranteed to be unique across all applications registered with the storage service.

szLuid

A null-terminated string which specifies a locally unique value associated with the registered application. This value is used internally by the storage service and guaranteed to be unique to the storage account that has registered the application. Note that there are no functions that currently accept the application LUID as parameter, but you may choose to use this value in your own application for other purposes.

dwTokens

An unsigned integer value which specifies the number of access tokens associated with the registered application. Typically there is only one access token associated with a given AppID at any one time, although it is possible that the API may allocate additional access tokens under some circumstances.

dwReserved

An unsigned integer value that is reserved for future use. This value will always be zero.

stCreated

A SYSTEMTIME structure that specifies the date and time the AppID was registered. This value is represented using Coordinated Universal Time (UTC) and is not adjusted for the local time zone.

stUpdated

A SYSTEMTIME structure that specifies the date and time the AppID object was last updated. This value is represented using Coordinated Universal Time (UTC) and is not adjusted for the local time zone. When a storage object is first created, this value will be the same as the object creation time.

Remarks

This structure is used in conjunction with the **GetFirstApplication** and **GetNextApplication**

methods which enumerate all registered applications for the current storage account.

To adjust the creation and update times to account for the local time zone, use the **SystemTimeToTzSpecificLocalTime** function. If you prefer to use FILETIME values, use the **SystemTimeToFileTime** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

WEB_STORAGE_OBJECT Structure

The **WEB_STORAGE_OBJECT** structure contains information about an individual storage object.

```
typedef struct _WEB_STORAGE_OBJECT
{
    TCHAR        szObjectId[64];
    TCHAR        szLabel[512];
    TCHAR        szDigest[128];
    TCHAR        szContent[128];
    DWORD        dwAttributes;
    DWORD        dwObjectSize;
    SYSTEMTIME   stCreated;
    SYSTEMTIME   stModified;
} WEB_STORAGE_OBJECT, *WEB_STORAGE_OBJECT;
```

Members

szObjectId

A null-terminated string which contains the unique identifier associated with this object. Object IDs are guaranteed to be unique for each storage object that is created by the application. The maximum length of an object ID is 64 characters, including the terminating null character.

szLabel

A null-terminated string which contains the label assigned to the object by the application. Object labels are case-sensitive and must be unique for each object. An application uses labels to reference an object with a human-recognizable name, rather than referencing them by their object ID. The maximum length of an object label is 512 characters, including the terminating null character.

szDigest

A null-terminated string which specifies the digest of the object contents, computed using an SHA-256 hash. The maximum length of the *szDigest* string is 128 characters, including the terminating null character. However, the digest value is always represented as a string of hexadecimal numbers that is exactly 64 characters long. It is important to note that even a zero-length object will have a digest, which is the standard SHA-256 NULL hash value.

szContent

A null-terminated string which specifies the MIME content type for the storage object. The content type is determined by the object label and evaluating the contents of the object. It is also possible for the application to explicitly specify the content type of the object when it is created.

dwAttributes

An unsigned integer value that specifies the attributes for the storage object. The object attributes are comprised of one or more bitflags:

Value	Constant	Description
0	WEB_OBJECT_DEFAULT	Default object attributes. This value is used to indicate the object can be modified, or that the attributes for a previously existing object should not be changed.
1	WEB_OBJECT_NORMAL	A normal object that that can be read and modified by the application. This is the default

		attribute for new objects that are created by the application.
2	WEB_OBJECT_READONLY	A read-only object that can only be read by the application. Attempts to modify or replace the contents of the object will fail. Read-only objects can be deleted.
4	WEB_OBJECT_HIDDEN	A hidden object. Objects with this attribute are not returned when enumerated using the EnumObjects method. The object can only be accessed directly when specifying its label.

dwObjectSize

An unsigned integer value that specifies the size of the storage object in bytes. The maximum size of an individual object is determined by the storage quota limits established for the account.

stCreated

A SYSTEMTIME structure that specifies the date and time the storage object was created. This value is represented using Coordinated Universal Time (UTC) and is not adjusted for the local time zone.

stModified

A SYSTEMTIME structure that specifies the date and time the storage object was last modified. This value is represented using Coordinated Universal Time (UTC) and is not adjusted for the local time zone. When a storage object is first created, this value will be the same as the object creation time.

Remarks

The object content type will always be in the format **type/subtype** where the type specifies a common media type (e.g.: text, audio, video, etc.) and subtype specifies the specific content. The most common content type for text files is **text/plain**. If the content type is unknown, the default content type is **application/octet-stream**.

Text objects may also optionally include the character encoding as part of the content type. For example, if an object contains UTF-8 encoded text, the content type may be returned as **text/plain; charset=utf-8**. If your application is parsing the content types, you must check if a character encoding was also included in the value. Text objects that do not specify an encoding either contain ASCII or text which uses the system code page. Unicode text will always be stored using UTF-8 encoding.

To adjust the object creation and modification times to account for the local time zone, use the **SystemTimeToTzSpecificLocalTime** method. If you prefer to use FILETIME values, use the **SystemTimeToFileTime** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

WEB_STORAGE_QUOTA Structure

The **WEB_STORAGE_QUOTA** structure contains information about the current storage account data usage and limits. This structure is used with the **GetStorageQuota** method.

```
typedef struct _WEB_STORAGE_QUOTA
{
    DWORD        dwObjects;
    DWORD        dwObjectLimit;
    DWORD        dwObjectSize;
    ULONGLONG    ulBytesUsed;
    ULONGLONG    ulBytesFree;
    ULONGLONG    ulStorageLimit;
} WEB_STORAGE_QUOTA, *LPWEB_STORAGE_QUOTA;
```

Members

dwObjects

An unsigned integer value which specifies the number of storage objects allocated for the account. This value may not exceed the total number of objects specified by the *dwObjectLimit* member.

dwObjectLimit

An unsigned integer value which specifies the maximum number of storage objects that may be created. In addition to the limit on the total amount of storage that may be used, there is a limit on the total number of objects that may be created by all applications.

dwObjectSize

An unsigned integer value which specifies the maximum size of an individual storage object. In addition to a limit on the total amount of storage used and the number of objects created, each object stored by the application cannot exceed this size.

ulBytesUsed

An unsigned 64-bit integer value which specifies the total number of bytes of data allocated for all storage objects. This value may not exceed the total number of bytes of storage available, which is specified by the *ulStorageLimit* member.

ulBytesFree

An unsigned 64-bit integer value which specifies the number of bytes available for the storage of new objects. This value reflects the total amount of available storage across all applications registered with the development account. If this value is zero, your storage account has reached its storage limit.

ulStorageLimit

An unsigned 64-bit integer value which specifies the maximum number of bytes of data storage available. This limit applies to all applications registered with the development account.

Remarks

Storage quota limits are assigned for each SocketTools development account. The **GetStorageQuota** method will populate this structure with information about the limits on your account. Accounts that are created with an evaluation license have much lower quota limits than a standard account and should be used for testing purposes only. After the evaluation period has ended, all objects stored using the evaluation license will be deleted.

These values do not represent limits on storage usage by a specific application. Quotas limits

apply to all applications that are registered with the development account, which is identified with the runtime license key used to initialize the class instance.

If your storage quota has been exceeded, either because the total number of objects or the total bytes of storage has reached their limit, your applications will be unable to create new objects. Your application can continue to access existing objects, regardless of your current quota limits.

To free storage space, use the **DeleteObject** method to delete individual storage objects that are no longer needed by your application, or use the **ResetStorage** method to delete all objects in a container.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cstools10.h`

See Also

[DeleteObject](#), [GetStorageQuota](#), [ResetStorage](#)

WEB_STORAGE_TRANSFER Structure

This structure is used by the **CWebStorage::GetTransferStatus** method to return information about a data transfer in progress.

```
typedef struct _WEB_STORAGE_TRANSFER
{
    TCHAR        szObjectLabel[512];
    DWORD        dwStorageType;
    DWORD        dwBytesTotal;
    DWORD        dwBytesCopied;
    DWORD        dwBytesPerSecond;
    DWORD        dwTimeElapsed;
    DWORD        dwTimeEstimated;
} WEB_STORAGE_TRANSFER, *LPWEB_STORAGE_TRANSFER
```

Members

szObjectLabel

A null-terminated string that specifies the label for the object that is being retrieved, created or replaced.

dwStorageType

An integer value that identifies the storage container type. This will be one of the following values:

Constant	Description
WEB_STORAGE_GLOBAL (1)	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN (2)	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
WEB_STORAGE_MACHINE (3)	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER (4)	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account

is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

dwBytesTotal

An unsigned integer which specifies the total number of bytes that will be transferred. If the object is being downloaded to the local host, this is the size of the stored object. If the data is being uploaded from the local host to be stored on the server, it is the size of the buffer or local file.

dwBytesCopied

An unsigned integer which specifies the total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the data transfer is completed. This is based on the average number of bytes transferred per second.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Whois Protocol Class Library

Request registration information for an Internet domain name.

Reference

- [Class Methods](#)
- [Error Codes](#)

Library Information

Class Name	CWhoisClient
File Name	CSWHOV10.DLL
Version	10.0.1468.2518
LibID	A1E831CF-F84F-48B1-9710-D012B50AB6D5
Import Library	CSWHOV10.LIB
Dependencies	None
Standards	RFC 954

Overview

The Whois protocol library provides an interface for requesting registration information for an Internet domain name. When a domain name is registered, the organization that registers the domain must provide certain contact information along with technical information such as the primary name servers for that domain. The library provides an API for requesting that information and returning it to the program so that it can be displayed or processed. This library would be most commonly used to query the server at **whois.internic.net** to obtain information about a specific Internet domain name or an administrative contact at that domain.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Whois Protocol Class Methods

Class	Description
CWhoisClient	Constructor which initializes the current instance of the class
~CWhoisClient	Destructor which releases resources allocated by the class
Method	Description
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified client handle to another thread
Cancel	Cancel the current blocking operation
Connect	Connect to the specified server
DetachHandle	Detach the handle for the current instance of this class
DisableEvents	Disable asynchronous event notification
DisableTrace	Disable logging of socket function calls to the trace log
Disconnect	Disconnect from the current server
EnableEvents	Enable asynchronous event notification
EnableTrace	Enable logging of socket function calls to a file
FreezeEvents	Suspend or resume event handling by the calling process
GetErrorString	Return a description for the specified error code
GetHandle	Return the client handle used by this instance of the class
GetLastError	Return the last error code
GetStatus	Return the current client status.
GetTimeout	Return the number of seconds until an operation times out
IsBlocking	Determine if the client is blocked, waiting for information
IsConnected	Determine if the client is connected to the server
IsInitialized	Determine if the class has been successfully initialized
IsReadable	Determine if there is data available to be read from the server
Read	Read data returned by the server
RegisterEvent	Register an event callback function
Search	Search for the specified record
SetLastError	Set the last error code
SetTimeout	Set the number of seconds until an operation times out
ShowError	Display a message box with a description of the specified error
WhoisEventProc	Callback method that processes events generated by the client

CWhoisClient::CWhoisClient Method

`CWhoisClient();`

The **CWhoisClient** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **WhoisInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cswhov10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[~CWhoisClient](#), [IsInitialized](#)

CWhoisClient::~CWhoisClient

`~CWhoisClient();`

The **CWhoisClient** destructor releases resources allocated by the current instance of the **CWhoisClient** object. It also uninitialized the library if there are no other concurrent uses of the class.

Remarks

When a **CWhoisClient** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the client session are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cswhov10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CWhoisClient](#)

CWhoisClient::AttachHandle Method

```
VOID AttachHandle(  
    HCLIENT hClient  
);
```

The **AttachHandle** method attaches the specified client handle to the current instance of the class.

Parameters

hClient

The handle to the client session that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a client handle created outside of the class using the SocketTools API. Once the client handle is attached to the class, the other class member functions may be used with that client session.

If a client handle already has been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous client session from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

Note that the *hClient* parameter is presumed to be a valid client handle and no checks are performed to ensure that the handle is valid. Specifying an invalid client handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[AttachThread](#), [DetachHandle](#), [GetHandle](#)

CWhoisClient::AttachThread Method

```
DWORD AttachThread(  
    DWORD dwThreadId  
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is WHOIS_ERROR. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[AttachHandle](#), [Cancel](#), [Connect](#), [DetachHandle](#), [Disconnect](#), [GetHandle](#)

CWhoisClient::Cancel Method

```
INT Cancel();
```

The **Cancel** method cancels any outstanding blocking operation in the client, causing the blocking method to fail. The application may then retry the operation or terminate the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is WHOIS_ERROR. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[IsBlocking](#)

CWhoisClient::Connect Method

```
BOOL Connect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwReserved,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **Connect** method establishes a connection with the specified server.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used. For standard connections, the default port number is 43.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwReserved

A reserved argument. This value should always be zero.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur. If this argument is NULL, then the client session will be blocking and no network events will be sent to the client.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the *hEventWnd* argument is NULL, this argument should be specified as WM_NULL.

Return Value

If the method succeeds, the return value is a non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The use of Windows event notification messages has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

To prevent this method from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **Connect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

If you specify an event notification window, then the client session will be asynchronous. When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
WHOIS_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
WHOIS_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
WHOIS_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
WHOIS_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
WHOIS_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
WHOIS_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To cancel asynchronous notification and return the client to a blocking mode, use the **DisableEvents** methods.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cswhov10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableEvents](#), [Disconnect](#), [EnableEvents](#), [RegisterEvent](#)

CWhoisClient::DetachHandle Method

```
HCLIENT DetachHandle();
```

The **DetachHandle** method detaches the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to detach a client handle created by the class for use with the SocketTools API. Once the client handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly released at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cswhov10.lib`

See Also

[AttachHandle](#), [GetHandle](#)

CWhoisClient::DisableEvents Method

```
INT DisableEvents();
```

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is WHOIS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the client has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[EnableEvents](#), [RegisterEvent](#)

CWhoisClient::DisableTrace Method

```
BOOL DisableTrace();
```

The **DisableTrace** method disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[EnableTrace](#)

CWhoisClient::Disconnect Method

VOID Disconnect();

The **Disconnect** method terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

None.

Return Value

None.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[Connect](#), [IsConnected](#)

CWhoisClient::EnableEvents Method

```
INT EnableEvents(  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is WHOIS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Constant	Description
WHOIS_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
WHOIS_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
WHOIS_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
WHOIS_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking

	operation. This event is only generated if the client is in asynchronous mode.
WHOIS_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
WHOIS_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[DisableEvents](#), [RegisterEvent](#)

CWhoisClient::EnableTrace Method

```
BOOL EnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **EnableTrace** method enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace method logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableTrace](#)

CWhoisClient::FreezeEvents Method

```
INT FreezeEvents(  
    BOOL bFreeze  
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the client.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is WHOIS_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DisableEvents](#), [EnableEvents](#), [RegisterEvent](#)

CWhoisClient::GetErrorString Method

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);  
  
INT GetErrorString(  
    DWORD dwErrorCode,  
    CString& strDescription  
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetLastError](#), [SetLastError](#), [ShowError](#)

CWhoisClient::GetHandle Method

```
HCLIENT GetHandle();
```

The **GetHandle** method returns the client handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active client session, the value `INVALID_CLIENT` will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cswhov10.lib`

See Also

[AttachHandle](#), [DetachHandle](#), [IsInitialized](#)

CWhoisClient::GetLastError Method

```
DWORD GetLastError();  
  
DWORD GetLastError(  
    CString& strDescription  
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError(0)** when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or WHOIS_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswhov10.lib

See Also

[GetErrorString](#), [SetLastError](#), [ShowError](#)

CWhoisClient::GetStatus Method

```
INT GetStatus();
```

The **GetStatus** method the current status of the client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the client status code. If the method fails, the return value is WHOIS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetStatus** method returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Constant	Description
0	WHOIS_STATUS_UNUSED	No connection has been established.
1	WHOIS_STATUS_IDLE	The client is current idle and not sending or receiving data.
2	WHOIS_STATUS_CONNECT	The client is establishing a connection with the server.
3	WHOIS_STATUS_READ	The client is reading data from the server.
4	WHOIS_STATUS_WRITE	The client is writing data to the server.
5	WHOIS_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified client session. To obtain status information about a file transfer, use the **GetTransferStatus** method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[IsBlocking](#), [IsInitialized](#), [IsReadable](#)

CWhoisClient::GetTimeout Method

```
INT GetTimeout();
```

The **GetTimeout** method returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is WHOIS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[Connect](#), [IsReadable](#), [Read](#), [SetTimeout](#)

CWhoisClient::IsBlocking Method

BOOL IsBlocking();

The **IsBlocking** method is used to determine if the client is currently performing a blocking operation.

Parameters

None.

Return Value

If the client is performing a blocking operation, the method returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the method returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking method may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IsBlocking** method can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Cancel](#), [GetStatus](#), [IsConnected](#), [IsInitialized](#), [IsReadable](#), [Read](#)

CWhoisClient::IsConnected Method

```
BOOL IsConnected();
```

The **IsConnected** method is used to determine if the client is currently connected to a server.

Parameters

None.

Return Value

If the client is connected to a server, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsInitialized](#), [IsReadable](#)

CWhoisClient::IsInitialized Method

```
BOOL IsInitialized();
```

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[CWhoisClient](#), [IsBlocking](#), [IsConnected](#)

CWhoisClient::IsReadable Method

```
BOOL IsReadable(  
    INT nTimeout,  
    LPDWORD LpdwAvail  
);
```

The **IsReadable** method is used to determine if data is available to be read from the server.

Parameters

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the method returns a non-zero value. If the client cannot read any data, the method returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsConnected](#), [IsInitialized](#)

CWhoisClient::Read Method

```
INT Read(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Read(  
    CString& strBuffer,  
    INT cbBuffer  
);
```

The **Read** method reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be copied. An alternate form of this method allows a **CString** variable to be passed and data read from the socket will be returned in that string.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the method fails, the return value is WHOIS_ERROR. To get extended error information, call **GetLastError**.

Remarks

When **Read** is called and the client is in non-blocking mode, it is possible that the method will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[EnableEvents](#), [IsBlocking](#), [IsReadable](#), [RegisterEvent](#)

CWhoisClient::RegisterEvent Method

```
INT RegisterEvent(  
    UINT nEventId,  
    WHOISEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Constant	Description
WHOIS_EVENT_CONNECT	The connection to the server has completed.
WHOIS_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
WHOIS_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
WHOIS_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
WHOIS_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
WHOIS_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

LpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **WhoisEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `WHOIS_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **WhoisEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a `DWORD_PTR` value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cswhov10.lib`

See Also

[DisableEvents](#), [EnableEvents](#), [FreezeEvents](#), [WhoisEventProc](#)

CWhoisClient::Search Method

```
INT Search(  
    LPCTSTR lpszKeyword,  
    UINT nSearchType  
);
```

The **Search** method submits the specified keyword to the server.

Parameters

lpszKeyword

Points to a string which specifies the query keyword. It may be a handle, name or mailbox.

nSearchType

The type of search being performed. One of the following values may be used:

Constant	Description
WHOIS_SEARCH_ANY	Search for any record that matches the specified keyword.
WHOIS_SEARCH_HANDLE	Search only for handles that match the specified keyword.
WHOIS_SEARCH_MAILBOX	Search only for mailboxes that match the specified keyword.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is WHOIS_ERROR. To get extended error information, call **GetLastError**.

Example

```
CWhoisClient whoisClient;  
  
// Connect to the WHOIS server at whois.internic.net  
if (whoisClient.Connect(_T("whois.internic.net")) == FALSE)  
{  
    whoisClient.ShowError();  
    return;  
}  
  
// Request information about the sockettools.com domain name  
if (whoisClient.Search(_T("sockettools.com")) == WHOIS_ERROR)  
{  
    whoisClient.ShowError();  
    whoisClient.Disconnect();  
    return;  
}  
  
// Read the response from the server, which contains information  
// about the domain and the registry that maintains the record  
CString strResult;  
CString strBuffer;  
INT nResult;  
  
do  
{  
    if ((nResult = whoisClient.Read(strBuffer)) > 0)  
        strResult += strBuffer;
```

```
}  
while (nResult > 0);  
  
// If there was an error reading the data, alert the user  
if (nResult == WHOIS_ERROR)  
    whoisClient.ShowError();  
  
whoisClient.Disconnect();
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Connect](#), [Disconnect](#), [IsReadable](#), [Read](#)

CWhoisClient::SetLastError Method

```
VOID SetLastError(  
    DWORD dwErrorCode  
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or WHOIS_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshov10.lib

See Also

[GetErrorString](#), [GetLastError](#), [ShowError](#)

CWhoisClient::SetTimeout Method

```
INT SetTimeout(  
    UINT nTimeout  
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is WHOIS_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[Connect](#), [GetTimeout](#), [IsReadable](#), [Read](#)

CWhoisClient::ShowError Method

```
INT ShowError(  
    LPCTSTR lpszAppTitle,  
    UINT uType,  
    DWORD dwErrorCode  
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A pointer to a string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[GetErrorString](#), [GetLastError](#), [SetLastError](#)

SocketTools Library Reference

- Domain Name Service Library
- File Encoding Library
- File Transfer Protocol Library
- File Transfer Server Library
- Hypertext Transfer Protocol Library
- Hypertext Transfer Server Library
- Internet Control Message Protocol Library
- Internet Message Access Protocol Library
- Mail Message Library
- Network News Protocol Library
- News Feed Library
- Post Office Protocol Library
- Remote Command Library
- Simple Mail Transfer Protocol Library
- Secure Shell Protocol Library
- SocketWrench Library
- Telnet Protocol Library
- Terminal Emulation Library
- Text Message Library
- Time Protocol Library
- Web Services Library
- Whois Protocol Library

Domain Name Service Library

Resolve domain names into Internet addresses and return information about a remote host, such as the servers that are responsible for accepting mail for the domain.

Reference

- [Functions](#)
- [Constants](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

File Name	CSDNSV10.DLL
Version	10.0.1468.2518
LibID	F247C1C3-4DB3-4DF2-A16E-88E2B97B5119
Import Library	CSDNSV10.LIB
Dependencies	None
Standards	RFC 1034

Overview

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

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Domain Name Service Functions

Function	Description
DnsAttachThread	Attach the specified client handle to another thread
DnsCancel	Cancel an outstanding nameserver query
DnsCloseHandle	Close the current client session
DnsCreateHandle	Create a new handle for the current client session
DnsDisableTrace	Disable logging of socket function calls to the trace log
DnsEnableTrace	Enable logging of socket function calls to a file
DnsEnumHostAliases	Enumerate the aliases for the specified host name or address
DnsEnumMailExchanges	Return a list of mail exchanges for the specified host name or IP address
DnsFormatAddress	Convert a numeric IPv4 address to a string
DnsGetAddress	Convert an IPv4 address string in dotted notation to a numeric IP address
DnsGetAddressFamily	Return the address family for the specified IP address
DnsGetDefaultHostFile	Return the fully qualified path name of the default host file on the local system
DnsGetErrorString	Return a description for the specified error code
DnsGetHostAddress	Return the IP address for the specified hostname
DnsGetHostByAddress	Return a pointer to data for the specified host IP address
DnsGetHostByName	Return a pointer to data for the specified host name
DnsGetHostFile	Return the name of the current host file
DnsGetHostInfo	Return additional information for the specified host
DnsGetHostName	Return the host name for the specified IP address
DnsGetHostServices	Return a list of services supported by the specified host
DnsGetLastError	Return the last error code
DnsGetLocalAddress	Return the IP address for the local host
DnsGetLocalDomain	Return the local domain name for the current session
DnsGetLocalName	Return the local host name
DnsGetMailExchange	Return the host that processes mail for the specified domain
DnsGetRecord	Return record data for the current host
DnsGetResolverAddress	Return address of last nameserver that resolved query
DnsGetResolverOptions	Return the current resolver options for the specified client session
DnsGetRetryCount	Get the number of times the query is sent to each server
DnsGetServerAddress	Return the address of the specified nameserver
DnsGetServerPort	Return the port of the specified nameserver
DnsGetTimeout	Get the number of seconds until a query times out
DnsHostNameToUnicode	Converts the canonical form of a host name to its Unicode version
DnsInitialize	Initialize the library and validate the specified license key at runtime

DnsMatchHostName	Match a host name against of list of addresses including wildcards
DnsNormalizeHostName	Return the canonical form of a host name
DnsRegisterServer	Add a nameserver address to the current session
DnsReset	Reset the current client state
DnsSetHostFile	Specify the name of an alternate file to use when resolving hostnames and IP addresses
DnsSetLastError	Set the last error code
DnsSetLocalDomain	Set the local domain name for the current session
DnsSetResolverOptions	Set the resolver options for the specified client session
DnsSetRetryCount	Set the number of times the query is sent to each server
DnsSetTimeout	Set the number of seconds until a query times out
DnsUninitialize	Terminates the use of the library
DnsUnregisterServer	Remove a nameserver address from the current session

DnsAttachThread Function

```
DWORD WINAPI DnsAttachThread(  
    HCLIENT hClient  
    DWORD dwThreadId  
);
```

The **DnsAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **DnsAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **DnsAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **DnsCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **DnsUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csdnsv10.lib

See Also

[DnsCancel](#), [DnsCloseHandle](#), [DnsCreateHandle](#), [DnsUninitialize](#)

DnsCancel Function

```
INT WINAPI DnsCancel(  
    HCLIENT hClient  
);
```

The **DnsCancel** function cancels any outstanding queries initiated by the client.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetHostByAddress](#), [DnsGetHostByName](#), [DnsGetHostInfo](#), [DnsGetHostServices](#),
[DnsGetMailExchange](#), [DnsGetRecord](#)

DnsCloseHandle Function

```
INT WINAPI DnsCloseHandle(  
    HCLIENT hClient  
);
```

The **DnsCloseHandle** function closes the handle and releases any memory allocated for the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsCreateHandle](#), [DnsUninitialize](#), [DnsUnregisterServer](#)

DnsCreateHandle Function

```
HCLIENT WINAPI DnsCreateHandle(  
    INT nTimeout,  
    INT nRetries,  
    DWORD dwOptions  
);
```

The **DnsCreateHandle** function creates a new handle which can be used to query the default nameservers configured for the local host.

Parameters

nTimeout

The number of seconds that the client will wait for a response before failing the query. The constant `DNS_TIMEOUT` can be used to specify the default timeout period of 15 seconds.

nRetries

The number of attempts the client will make to resolve a query after a timeout. The constant `DNS_RETRIES` can be used to specify the default retry count of 4.

dwOptions

An unsigned integer that specifies one or more resolver options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
<code>DNS_OPTION_NONE</code>	No additional resolver options specified. This is the default value, and it is recommended that most applications do not specify additional options unless the implications of doing so are understood.
<code>DNS_OPTION_PROMOTE</code>	Promotes the server that successfully completed the last query to the first server that will be used to resolve subsequent queries. This option can improve performance in some cases where one or more of the registered servers are non-responsive. This option takes precedence over the <code>DNS_OPTION_ROTATE</code> option.
<code>DNS_OPTION_ROTATE</code>	Enables a round-robin selection of nameservers when performing queries. Normally each nameserver is queried in the same order. This option rotates the available nameservers so a different server is used with each query.
<code>DNS_OPTION_AUTHONLY</code>	Require the answer from the nameserver to be authoritative, not from the server's cache. This option is included for future expansion as most servers do not support this feature and will ignore it.
<code>DNS_OPTION_PRIMARY</code>	Queries are only accepted from the primary nameserver. This option is included for future expansion as most servers do not support this feature and will ignore it.

DNS_OPTION_NORECURSE	Disable the sending of recursive queries to the nameserver. Specifying this option will disable the bit in the DNS request header that specifies recursion is desired.
DNS_OPTION_NOSEARCH	Disable additional queries of higher domains in the search list if the host name cannot be resolved. If this option is specified, and the host name cannot be resolved using the local domain name an error is returned immediately. This option is ignored if no local domain has been specified or if the DNS_OPTION_NOSUFFIX option has been specified.
DNS_OPTION_NOSUFFIX	Disable additional queries using the local domain name if the host name is not a fully qualified domain name and cannot be resolved. This option is ignored if no local domain has been specified.
DNS_OPTION_NONAMECHECK	Disable checking the host name for invalid characters, such as the underscore and control characters. By default, host names are checked to ensure they're valid before submitting a query to the nameserver.
DNS_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is `INVALID_CLIENT`. To get extended error information, call **DnsGetLastError**.

Remarks

The **DnsCreateHandle** function returns a handle to the application which can be used to perform nameserver queries. By default, the library will use the nameservers which are configured for the local host. To determine the IP addresses of those nameservers, use the **DnsGetServerAddress** function.

To specify alternate nameservers, either as replacements for or additions to the default nameservers, use the **DnsRegisterServer** function.

It is recommended that most applications do not specify any resolver options and use the default behavior. Specifying these options without understanding how they can affect standard queries can result in unexpected failures. In particular, caution should be used when specifying the `DNS_OPTION_NORECURSE` and `DNS_OPTION_NOSEARCH` options as they change the normal process of resolving a host name.

Specifying the `DNS_OPTION_FREETHREAD` option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same

handle.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

See Also

[DnsCloseHandle](#), [DnsGetResolverOptions](#), [DnsGetServerAddress](#), [DnsInitialize](#), [DnsRegisterServer](#),
[DnsSetLocalDomain](#), [DnsSetResolverOptions](#)

DnsDisableTrace Function

```
BOOL WINAPI DnsDisableTrace();
```

The **DnsDisableTrace** function disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsEnableTrace](#)

DnsEnableTrace Function

```
BOOL WINAPI DnsEnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **DnsEnableTrace** function enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the remote host.

Trace function logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsDisableTrace](#)

DnsEnumHostAliases Function

```
INT WINAPI DnsEnumHostAliases(  
    HCLIENT hClient,  
    LPCTSTR lpszHostName,  
    LPTSTR *lpszHostAlias,  
    INT nMaxAliases  
);
```

The **DnsEnumHostAliases** function returns a list of aliases for the specified host name or IP address.

Parameters

hClient

Handle to the client session.

lpszHostName

Pointer to the string that contains the hostname to be resolved. If a fully qualified domain name is not provided, the default local domain will be used. This value may also be an IP address.

lpszHostAlias

Pointer to an array of string pointers which specify one or more host aliases. If the application needs to store these values, a local copy should be made because they are invalidated when another host name is resolved.

nMaxAliases

The maximum number of aliases in the array. This parameter must have a value of at least one, or an error will be returned.

Return Value

If the function succeeds, the return value is the number of host aliases. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The application must never attempt to modify the host aliases or free any of the values. This function uses an internal data structure to store the host information and only one copy of this structure is allocated per thread. The application must copy any information it needs before issuing any other function calls.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetHostAddress](#), [DnsGetHostName](#)

DnsEnumMailExchanges Function

```
INT WINAPI DnsEnumMailExchanges(  
    HCLIENT hClient,  
    LPCTSTR lpszHostName,  
    LPTSTR *lpszMailExchanges,  
    INT nMaxMailExchanges  
);
```

The **DnsEnumMailExchanges** function returns a list of mail exchanges for the specified host name or IP address.

Parameters

hClient

Handle to the client session.

lpszHostName

Pointer to the string that contains the hostname or domain name to be queried.

lpszMailExchanges

Pointer to an array of string pointers which specify one or more mail exchanges. If the application needs to store these values, a local copy should be made because they are invalidated when another host name is resolved. The list of mail exchange records is sorted in priority order, from highest (i.e., those whose preference value is smallest) to lowest.

nMaxMailExchanges

The maximum number of mail exchanges in the array. If this parameter is 0, then the function will return the number of mail exchanges, but the list of mail exchanges will not be output in *lpszMailExchanges*.

Return Value

If the function succeeds, the return value is the number of mail exchanges. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The application must never attempt to modify the mail exchange host names or free any of the values. This function uses an internal data structure to store the host information and only one copy of this structure is allocated per thread. The application must copy any information it needs before issuing any other function calls.

Example

```
// Count the number of mail exchanges  
if ((nMX = DnsEnumMailExchanges(hClient,szHostName, NULL, 0)) == DNS_ERROR)  
{  
    dwError = DnsGetLastError();  
    DnsGetErrorString(dwError, szError, BUFSIZE);  
    printf("DnsEnumMailExchanges failed for %s: %s\n", szHostName, szError);  
}  
else  
{  
    int nIndex;  
    printf("%d MX for %s\n",nMX, szHostName);  
  
    // Allocate memory for the list of mail exchanges
```

```
lpszMailExchanges = (LPTSTR *)malloc(nMX * sizeof(LPTSTR));

// Retrieve the list of mail exchanges
nMX = DnsEnumMailExchanges(hClient, szHostName, lpszMailExchanges, nMX);

for (nIndex = 0; nIndex < nMX; nIndex++)
    printf("#%d: %s\n", nIndex+1, *lpszMailExchanges++);

free(lpszMailExchanges);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetMailExchange](#)

DnsFormatAddress Function

```
INT WINAPI DnsFormatAddress(  
    LPINTERNET_ADDRESS lpAddress,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

The **DnsFormatAddress** function converts a numeric IPv4 or IPv6 address to a string.

Parameters

lpAddress

A pointer to an INTERNET_ADDRESS structure which specifies the numeric IPv4 or IPv6 address to be converted into a string.

lpszAddress

A pointer to a null-terminated array of characters which will contain the converted IPv4 address in dot-notation. This string should be at least 16 characters in length.

nMaxLength

The maximum number of characters which may be copied in to the string buffer.

Return Value

If the function succeeds, the return value is the length of the string buffer. If the function fails, the return value is zero. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetAddress](#), [DnsGetHostAddress](#), [DnsGetHostInfo](#), [DnsGetHostServices](#), [DnsGetMailExchange](#), [DnsGetRecord](#), [INTERNET_ADDRESS](#)

DnsGetAddress Function

```
INT WINAPI DnsGetAddress(  
    LPCTSTR lpszAddress,  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddress  
);
```

The **DnsGetAddress** function converts an address string to a numeric IPv4 or IPv6 address.

Parameters

lpszAddress

A pointer to a string which specifies an IPv4 address in dotted notation.

nAddressFamily

An integer value which specifies the type of IP address. If this parameter is zero, the address family will be determined automatically based on the format of the address string. If this parameter is `DNS_ADDRESS_IPV4`, the address must be in IPv4 format, and if it is `DNS_ADDRESS_IPV6`, the address must be in IPv6 format.

lpAddress

A pointer to an `INTERNET_ADDRESS` structure that will contain the numeric form of the IPv4 or IPv6 address in network byte order when the function returns.

Return Value

If the function succeeds, the return value is the address family for the IP address. If the function fails, the return value is `DNS_ERROR`. To get extended error information, call **DnsGetLastError**.

Remarks

This function will only accept a string that is in the proper format for an IP address, and cannot be used to resolve a host name. To perform host name resolution, use the **DnsGetHostAddress** function. To convert a numeric address to an address string, use the **DnsFormatAddress** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsFormatAddress](#), [DnsGetHostAddress](#), [DnsGetHostInfo](#), [DnsGetHostServices](#),
[DnsGetMailExchange](#), [DnsGetRecord](#), [INTERNET_ADDRESS](#)

DnsGetAddressFamily Function

```
INT WINAPI DnsGetAddressFamily(  
    LPCTSTR LpszAddress  
);
```

The **DnsGetAddressFamily** function returns the address family for the specified IP address.

Parameters

lpszAddress

A pointer to a string which specifies an IPv4 or IPv6 address.

Return Value

If the function succeeds, the return value is DNS_ADDRESS_IPV4 if the address is in IPv4 format, or DNS_ADDRESS_IPV6 if the address is in IPv6 format. If the address string is not in a recognized format, it returns DNS_ADDRESS_ANY.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsFormatAddress](#), [DnsGetHostAddress](#), [DnsGetHostInfo](#), [DnsGetHostServices](#),
[DnsGetMailExchange](#), [DnsGetRecord](#)

DnsGetDefaultHostFile Function

```
INT WINAPI DnsGetDefaultHostFile(  
    LPTSTR lpszFileName,  
    INT nMaxLength  
);
```

The **DnsGetDefaultHostFile** function returns the fully qualified path name of the host file on the local system. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **DnsGetHostAddress** and **DnsGetHostName** function. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

```
ipaddress hostname [hostalias ...]
```

For example, one typical entry maps the name "localhost" to the local loopback IP address. This would be entered as:

```
127.0.0.1 localhost
```

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

The default hosts file is stored in C:\Windows\system32\drivers\etc\hosts and may or may not exist on a given system. Note that there is no extension for this file.

Parameters

lpszFileName

Pointer to a string buffer that will contain the fully qualified file name to the default host file. It is recommended that this buffer be at least MAX_PATH characters in size. This parameter may be NULL, in which case the function will return the length of the string, not including the terminating null byte.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the function succeeds, the return value is length of the string. A return value of zero indicates that the default host file could not be determined for the current platform. To get extended error information, call **DnsGetLastError**.

Remarks

This function only returns the default location of the host file and does not determine if the file actually exists. It is not required that a host file be present on the system.

The default host file is processed before performing a nameserver lookup when resolving a hostname into an IP address, or an IP address into a hostname.

To specify an alternate local host file, use the **DnsSetHostFile** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csdns10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetHostAddress](#), [DnsGetHostFile](#), [DnsGetHostName](#), [DnsSetHostFile](#)

DnsGetErrorString Function

```
INT WINAPI DnsGetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT nMaxLength  
);
```

The **DnsGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **DnsGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The last-error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is 0, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetLastError](#), [DnsSetLastError](#)

DnsGetHostAddress Function

```
INT WINAPI DnsGetHostAddress(  
    HCLIENT hClient,  
    INT nAddressFamily,  
    LPCTSTR lpszHostName,  
    LPTSTR lpszHostAddress,  
    INT nMaxLength  
);
```

The **DnsGetHostAddress** function resolves the specified host name, storing the IP address in the provided buffer.

Parameters

hClient

Handle to the client session.

nAddressFamily

An integer value which specifies the type of address that should be returned. A value of `DNS_ADDRESS_IPV4` specifies that the IPv4 address for the host should be returned. A value of `DNS_ADDRESS_IPV6` specifies that the IPv6 address for the host should be returned. A value of `DNS_ADDRESS_ANY` specifies that if the host only has an IPv6 address, that value should be returned, otherwise return the IPv4 address for the host.

lpszHostName

Pointer to the string that contains the hostname to be resolved. If a fully qualified domain name is not provided, the default local domain will be used.

lpszHostAddress

Pointer to the buffer that will contain the IP address, stored as a string in dot notation. This buffer should be at least 48 characters in length to accommodate both IPv4 and IPv6 addresses. The format of the address is determined by the address family specified.

nMaxLength

The maximum length of the string buffer. The maximum length of the buffer must include the terminating null character.

Return Value

If the function succeeds, the return value is the number of characters copied into the host address buffer. If the function fails, the return value is `DNS_ERROR`. To get extended error information, call **DnsGetLastError**.

Remarks

The **DnsGetHostAddress** function may return an address in either IPv4 or IPv6 format, depending on the address family that is specified and what records exist for the host. If your application does not support the IPv6 address format, you must specify the *nAddressFamily* parameter as `DNS_ADDRESS_IPV4` to prevent the possibility of an IPv6 address being returned.

If the *nAddressFamily* parameter is specified as `DNS_ADDRESS_ANY`, this function will first check for an IPv4 address record for the host. If it exists, it will return that address. If the host does not have an IPv4 address, it will then check for an IPv6 address record and return that address. This gives preference to IPv4 addresses, but your application should never depend on this behavior. In the future, this function may change to give preference to IPv6 addresses.

To determine what format an address is in, use the **DnsGetAddressFamily** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsEnumHostAliases](#), [DnsGetAddressFamily](#), [DnsGetHostName](#), [DnsGetMailExchange](#),
[DnsGetRecord](#)

DnsGetHostByAddress Function

```
LPHOSTENT WINAPI DnsGetHostByAddress(  
    HCLIENT hClient,  
    LPVOID lpvAddress,  
    INT cbAddress,  
    INT nAddressFamily  
);
```

The **DnsGetHostByAddress** function returns a pointer to a [HOSTENT](#) structure which contains the results of a successful search for the host specified by address parameter.

Parameters

hClient

Handle to the client session.

lpvAddress

Pointer to an integer IPv4 address in network byte order.

cbAddress

The length of the address in bytes; this value should always be 4.

nAddressFamily

The type of address being resolved; this value should always be `DNS_ADDRESS_IPV4` as defined in the Windows Sockets header file.

Return Value

If the function succeeds, the return value is a pointer to a `HOSTENT` structure. If the function fails, the return value is `NULL`. To get extended error information, call **DnsGetLastError**.

Remarks

The application must never attempt to modify this structure or to free any of its components. Only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other function calls. To convert an IPv4 address string in dotted notation to a 32-bit IP address, use the **DnsGetAddress** function.

This function is included for compatibility with existing applications which already use the `HOSTENT` structure. Because this function returns a pointer to a complex structure, it may not be suitable for some programming languages.

This function is not compatible with IPv6 addresses. For applications that must support both IPv4 and IPv6 address formats, use the **DnsGetHostAddress** and **DnsGetHostName** functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetHostAddress](#), [DnsGetHostByName](#), [DnsGetHostInfo](#), [DnsGetHostName](#), [DnsGetHostServices](#), [DnsGetRecord](#), [DnsGetResolverAddress](#), [DnsRegisterServer](#)

DnsGetHostByName Function

```
LPHOSTENT WINAPI DnsGetHostByName(  
    HCLIENT hClient,  
    LPCTSTR lpszHostName  
);
```

The **DnsGetHostByName** function returns a pointer to a [HOSTENT](#) structure which contains the results of a successful search for the host specified in the name parameter.

Parameters

hClient

Handle to the client session.

lpszHostName

Pointer to the string that contains the hostname to be resolved. If a fully qualified domain name is not provided, the default local domain will be used.

Return Value

If the function succeeds, the return value is a pointer to a [HOSTENT](#) structure. If the function fails, the return value is NULL. To get extended error information, call **DnsGetLastError**.

Remarks

The application must never attempt to modify this structure or to free any of its components. Only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other function calls. This function will automatically resolve an IP address passed as a string, converting it to numeric form and calling the **DnsGetHostByAddress** function.

This function is included for compatibility with existing applications which already use the [HOSTENT](#) structure. Because this function returns a pointer to a complex structure, it may not be suitable for some programming languages.

This function is not compatible with IPv6 addresses. For applications that must support both IPv4 and IPv6 address formats, use the **DnsGetHostAddress** and **DnsGetHostName** functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetHostAddress](#), [DnsGetHostByAddress](#), [DnsGetHostInfo](#), [DnsGetHostName](#),
[DnsGetHostServices](#), [DnsGetMailExchange](#), [DnsGetRecord](#), [DnsGetResolverAddress](#),
[DnsRegisterServer](#)

DnsGetHostFile Function

```
INT WINAPI DnsGetHostFile(  
    HCLIENT hClient,  
    LPTSTR lpszFileName,  
    INT nMaxLength  
);
```

The **DnsGetHostFile** function returns the name of the host file previously set using the **DnsSetHostFile** function. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **DnsGetHostAddress** and **DnsGetHostName** function.

Parameters

hClient

Handle to the client session.

lpszFileName

Pointer to a string buffer that will contain the host file name. It is recommended that this buffer be at least MAX_PATH characters in size. This parameter may be NULL, in which case the function will return the length of the string, not including the terminating null character.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the function succeeds, the return value is length of the string. A return value of zero indicates that no host file has been specified or the function was unable to determine the file name. To get extended error information, call **DnsGetLastError**. If the last error is zero, this indicates that no host file name has been specified for the current thread. If the last error is non-zero, this indicates the reason that the function failed.

Remarks

This function only returns the name of the host file that is cached in memory for the current thread. The contents of the file on the disk may have changed after the file was loaded into memory. To reload the host file or clear the cache, call the **DnsSetHostFile** function.

If a host file has been specified, it is processed before the default host file when resolving a hostname into an IP address, or an IP address into a hostname. If the host name or address is not found, or no host file has been specified, a nameserver lookup is performed.

The host file returned by this function may be different than the default host file for the local system. To determine the file name for the default host file, use the **DnsGetDefaultHostFile** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetDefaultHostFile](#), [DnsGetHostAddress](#), [DnsGetHostName](#), [DnsSetHostFile](#)

DnsGetHostInfo Function

```
INT WINAPI DnsGetHostInfo(  
    HCLIENT hClient,  
    LPCTSTR lpszHostName,  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);
```

The **DnsGetHostInfo** function returns the HINFO record for the specified hostname. This information, if it is provided, typically specifies the operating system type and hardware platform.

Parameters

hClient

Handle to the client session.

lpszHostName

Pointer to the string which specifies the host name that information will be returned for.

lpszBuffer

Pointer to the buffer which will contain the host information returned by the nameserver.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the function succeeds, the length of the host information buffer is returned. A return value of zero indicates that no information is available for the specified host. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

Many systems do not maintain HINFO records for a site since that information can potentially be used to compromise system security. The information is typically used for administrative purposes with internal networks.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetHostByAddress](#), [DnsGetHostByName](#), [DnsGetHostServices](#), [DnsGetMailExchange](#), [DnsGetRecord](#), [DnsGetResolverAddress](#), [DnsRegisterServer](#)

DnsGetHostName Function

```
INT WINAPI DnsGetHostName(  
    HCLIENT hClient,  
    LPCTSTR lpszHostAddress,  
    LPTSTR lpszHostName,  
    INT nMaxLength  
);
```

The **DnsGetHostName** function resolves the specified IP address, storing the fully qualified host name in the provided buffer.

Parameters

hClient

Handle to the client session.

lpszHostAddress

Pointer to a string that specifies an IPv4 or IPv6 formatted address.

lpszHostName

Pointer to the buffer that will contain the fully qualified domain name for the specified host. This buffer should be at least 64 characters in length.

nMaxLength

The maximum length of the string buffer.

Return Value

If the function succeeds, the return value is the number of characters copied into the host name buffer. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The **DnsGetHostName** function first looks to see if there is an entry in the local host file for the specified IP address, and if one exists, it will return the host name for that address. If you do not want to use the local host file at all, and only return an host name if a DNS query resolves the address, use the **DnsGetRecord** function and specify a record type of DNS_RECORD_PTR.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsEnumHostAliases](#), [DnsGetHostByAddress](#), [DnsGetHostByName](#), [DnsGetHostAddress](#)

DnsGetHostServices Function

```
INT WINAPI DnsGetHostServices(  
    HCLIENT hClient,  
    LPCTSTR lpszHostName,  
    INT nProtocol,  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);
```

The **DnsGetHostServices** function returns the WKS (Well Known Services) record for the specified hostname and protocol. This information, if it is provided, typically specifies the names of those services supported on the host.

Parameters

hClient

Handle to the client session.

lpszHostName

Pointer to the string which specifies the host name that information will be returned for.

nProtocol

The protocol for those services that information should be returned about. The following protocols are recognized:

Value	Constant	Description
6	DNS_PROTOCOL_TCP	Services that use the Transmission Control Protocol (TCP)
17	DNS_PROTOCOL_UDP	Services that use the User Datagram Protocol (UDP)

lpszBuffer

Pointer to the buffer which will contain the host information returned by the nameserver.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the function succeeds, the length of the host services buffer is returned. A return value of zero indicates that no information is available for the specified host. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

Many systems do not maintain complete services records for a site since that information can potentially be used to compromise system security. An application should not depend on this information being available for any given record.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetHostByAddress](#), [DnsGetHostByName](#), [DnsGetHostInfo](#), [DnsGetMailExchange](#),
[DnsGetRecord](#), [DnsGetResolverAddress](#), [DnsRegisterServer](#)

DnsGetLastError Function

```
DWORD WINAPI DnsGetLastError();
```

Parameters

None.

Return Value

The return value is the calling thread's last-error code value. Functions set this value by calling the **DnsSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last-error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **DnsGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **DnsSetLastError(0)** when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or DNS_ERROR. Those functions which call **DnsSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

See Also

[DnsGetErrorString](#), [DnsSetLastError](#)

DnsGetLocalAddress Function

```
INT WINAPI DnsGetLocalAddress(  
    HCLIENT hClient,  
    INT nAddressFamily,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

The **DnsGetLocalAddress** function returns the IP address for the local host.

hClient

Handle to the client session.

nAddressFamily

An integer value which specifies the type of address that should be returned. A value of `DNS_ADDRESS_IPV4` specifies that the IPv4 address for the host should be returned. A value of `DNS_ADDRESS_IPV6` specifies that the IPv6 address for the host should be returned. A value of `DNS_ADDRESS_ANY` specifies that if the host only has an IPv6 address, that value should be returned, otherwise return the IPv4 address for the host.

lpszAddress

Pointer to the buffer that will contain the IP address, stored as a string in dot notation. This buffer should be at least 40 characters in length to accommodate both IPv4 and IPv6 addresses.

nMaxLength

The maximum length of the string buffer.

Return Value

If the function succeeds, the return value is the number of characters copied into the host address buffer. If the function fails, the return value is `DNS_ERROR`. To get extended error information, call **DnsGetLastError**.

Remarks

The **DnsGetLocalAddress** function may return an address in either IPv4 or IPv6 format, depending on the address family that is specified and what records exist for the host. If your application does not support the IPv6 address format, you must specify the *nAddressFamily* parameter as `DNS_ADDRESS_IPV4` to prevent the possibility of an IPv6 address being returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetHostName](#), [DnsGetHostInfo](#), [DnsGetHostServices](#), [DnsGetMailExchange](#), [DnsGetRecord](#)

DnsGetLocalDomain Function

```
INT WINAPI DnsGetLocalDomain(  
    HCLIENT hClient,  
    LPTSTR lpszDomain,  
    INT nMaxLength  
);
```

The **DnsGetLocalDomain** function copies the local domain name into the specified buffer. The value returned is the same value that was set with the **DnsSetLocalDomain** function. If no local domain name has been set, an empty string is returned.

Parameters

hClient

Handle to the client session.

lpszDomain

Pointer to the buffer that is used to store the local domain name. If no local domain name has been set, this buffer will be set to zero length.

nMaxLength

The maximum number of bytes to copy into the buffer, including the null character terminator.

Return Value

If the function succeeds, the return value is the length of the domain name string. A return value of zero indicates that no local domain name has been set. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsRegisterServer](#), [DnsSetLocalDomain](#)

DnsGetLocalName Function

```
INT WINAPI DnsGetLocalName(  
    HCLIENT hClient,  
    LPTSTR lpszLocalName,  
    INT nMaxLength  
);
```

The **DnsGetLocalName** function returns the local host name.

Parameters

hClient

Handle to the client session.

lpszLocalName

Pointer to a string buffer that will contain the local host name. It is recommended that this buffer be at least 64 characters in size.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the function succeeds, the return value is the length of the local host name. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetHostByAddress](#), [DnsGetHostByName](#), [DnsGetHostInfo](#), [DnsGetHostServices](#),
[DnsGetMailExchange](#), [DnsGetRecord](#)

DnsGetMailExchange Function

```
INT WINAPI DnsGetMailExchange(  
    HCLIENT hClient,  
    LPCTSTR lpszHostName,  
    LPINT lpnPreference,  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);
```

The **DnsGetMailExchange** function returns the mail exchange (MX) record information for the specified domain. This information, if it is provided, identifies a server responsible for processing mail for the given domain.

Parameters

hClient

Handle to the client session.

lpszHostName

Pointer to the string which specifies the host name that information will be returned for.

lpnPreference

Pointer to the integer which will contain the preference for the specified mail exchange host.

lpszBuffer

Pointer to the buffer which will contain the host information returned by the nameserver.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the function succeeds, the length of the buffer is returned. A return value of zero indicates that no information is available for the specified host. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The mail exchange record is typically used by mail delivery agents to determine what system is responsible for accepting mail addressed to a given domain. This function will return the first MX record provided by the server. Note that some domains may have multiple mail servers. To enumerate all of the mail exchange records for a domain, use the **DnsEnumMailExchanges** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetHostByAddress](#), [DnsGetHostByName](#), [DnsGetHostInfo](#), [DnsGetHostServices](#), [DnsGetRecord](#), [DnsGetResolverAddress](#), [DnsRegisterServer](#), [DnsEnumMailExchanges](#)

DnsGetRecord Function

```
INT WINAPI DnsGetRecord(  
    HCLIENT hClient,  
    LPCTSTR lpszHostName,  
    INT nRecordType,  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);
```

The **DnsGetRecord** function returns the specified record information for the given hostname.

Parameters

hClient

Handle to the client session.

lpszHostName

Pointer to the string which specifies the host name that information will be returned for.

nRecordType

The record type for the information that should be returned. The following record types are recognized:

Value	Constant	Description
0	DNS_RECORD_NONE	No record type
1	DNS_RECORD_ADDRESS	IPv4 host address
2	DNS_RECORD_NS	Authoritative nameserver
5	DNS_RECORD_CNAME	Canonical host name (alias)
6	DNS_RECORD_SOA	Start of Authority
11	DNS_RECORD_WKS	Well known services
12	DNS_RECORD_PTR	Domain name
13	DNS_RECORD_HINFO	Host information
14	DNS_RECORD_MINFO	Mailbox information
15	DNS_RECORD_MX	Mail exchange host
16	DNS_RECORD_TXT	Text strings
28	DNS_RECORD_AAAA	IPv6 host address
29	DNS_RECORD_LOC	Location information
100	DNS_RECORD_UINFO	User information
101	DNS_RECORD_UID	User ID
102	DNS_RECORD_GID	Group ID

lpszBuffer

Pointer to the buffer which will contain the host information returned by the nameserver.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the function succeeds, the length of the information buffer is returned. A return value of zero indicates that no information for that record is available for the specified host. If the function fails, the return value is `DNS_ERROR`. To get extended error information, call **DnsGetLastError**.

Remarks

The **DnsGetRecord** function can be used to resolve a host name into an IPv4 address using the record type `DNS_RECORD_ADDRESS`, or an IPv6 address using the record type `DNS_RECORD_AAAA`. It can also be used to perform a reverse lookup and resolve an IP address into a host name by using the record type `DNS_RECORD_PTR`.

To determine the host that serves as the primary or master DNS for a zone, the record name should be specified as the domain name (e.g.: `microsoft.com`) and the record type should be `DNS_RECORD_SOA`. The value returned will be the fully qualified domain name for host.

Note that this function does not reference a local host file when resolving host names or addresses. If the record lookup fails, this function will return an error even if there's an entry for the host in the file that has been specified by a call to **DnsSetHostFile**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csdnsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetHostAddress](#), [DnsGetHostName](#), [DnsGetHostInfo](#), [DnsGetHostServices](#), [DnsGetMailExchange](#), [DnsGetResolverAddress](#), [DnsRegisterServer](#)

DnsGetResolverAddress Function

```
INT WINAPI DnsGetResolverAddress(  
    HCLIENT hClient,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

The **DnsGetResolverAddress** returns the address of the nameserver that resolved the last query.

Parameters

hClient

Handle to the client session.

lpszAddress

A pointer to a string buffer that will contain the address of the nameserver when the function returns. This buffer should be large enough to store both IPv4 and IPv6 addresses, with a minimum length of 40 characters. If this parameter is NULL, it will be ignored.

nMaxLength

The maximum number of characters that can be copied into the string buffer. If this value is zero, the *lpszAddress* parameter will be ignored and the function will return the length of the address.

Return Value

If the function succeeds, the return value is the length of the address, not including the terminating null character. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetHostByAddress](#), [DnsGetHostByName](#), [DnsGetHostInfo](#), [DnsGetHostServices](#),
[DnsGetMailExchange](#), [DnsGetRecord](#), [DnsRegisterServer](#)

DnsGetResolverOptions Function

```
DWORD WINAPI DnsGetResolverOptions(  
    HCLIENT hClient  
);
```

The **DnsGetResolverOptions** function returns the options that have been set for the client session.

Parameters

hClient

Handle to the client session

Return Value

If the function succeeds, the return value is the resolver options set for the client session. If the client handle is invalid or no resolver options have been specified, the function will return zero.

Remarks

The **DnsGetResolverOptions** function can be used to determine which resolver options have been specified for the client session. For a list of the available options, refer to the documentation for the **DnsSetResolverOptions** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

See Also

[DnsCreateHandle](#), [DnsSetResolverOptions](#)

DnsGetRetryCount Function

```
INT WINAPI DnsGetRetryCount(  
    HCLIENT hClient  
);
```

The **DnsGetRetryCount** returns the retry count for the current client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the retry count. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The retry count determines the amount of time the client will wait for a response from each query, the effective amount of time the client will wait increases with each nameserver and the total number of retries specified. For example, two nameservers registered with the client, with a default of 4 retries per nameserver and a timeout value of 10 seconds, would cause the client to wait a total of 80 seconds until it returns an error indicating that it was unable to resolve the query.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetTimeout](#), [DnsSetRetryCount](#), [DnsSetTimeout](#)

DnsGetServerAddress Function

```
INT WINAPI DnsGetServerAddress(  
    HCLIENT hClient,  
    INT nServer,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

The **DnsGetServerAddress** function returns the address of the registered nameserver.

Parameters

hClient

Handle the client session.

nServer

The index into the client's nameserver table. This index is the same value that is passed to the **DnsRegisterServer** function when the nameserver is registered.

lpszAddress

A pointer to a string buffer that will contain the address of the nameserver when the function returns. This buffer should be large enough to store both IPv4 and IPv6 addresses, with a minimum length of 40 characters. If this parameter is NULL, it will be ignored.

nMaxLength

The maximum number of characters that can be copied into the string buffer. If this value is zero, the *lpszAddress* parameter will be ignored and the function will return the length of the address.

Return Value

If the function succeeds, the return value is the length of the address, not including the terminating null character. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsCreateHandle](#), [DnsGetResolverAddress](#), [DnsGetServerPort](#), [DnsRegisterServer](#),
[DnsUnregisterServer](#)

DnsGetServerPort Function

```
INT WINAPI DnsGetServerPort(  
    HCLIENT hClient,  
    INT nServer  
);
```

The **DnsGetServerPort** function returns the port number registered to the specified nameserver.

Parameters

hClient

Handle the client session.

nServer

The index into the client's nameserver table. This index is the same value that is passed to the **DnsRegisterServer** function when the nameserver is registered.

Return Value

If the function succeeds, the return value is the specified port number. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetResolverAddress](#), [DnsGetServerAddress](#), [DnsRegisterServer](#), [DnsUnregisterServer](#)

DnsGetTimeout Function

```
INT WINAPI DnsGetTimeout(  
    HCLIENT hClient  
);
```

The **DnsGetTimeout** function returns the timeout value for the current client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The timeout value determines the amount of time the client will wait for a response from each query, the effective amount of time the client will wait increases with each nameserver and the total number of retries specified. For example, two nameservers registered with the client, with a default of 4 retries per nameserver and a timeout value of 10 seconds, would cause the client to wait a total of 80 seconds until it returns an error indicating that it was unable to resolve the query.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

See Also

[DnsGetRetryCount](#), [DnsSetRetryCount](#), [DnsSetTimeout](#)

DnsHostNameToUnicode Function

```
INT WINAPI DnsHostNameToUnicode(  
    LPCTSTR lpszHostName,  
    LPTSTR lpszUnicodeName,  
    INT nMaxLength  
);
```

The **DnsHostNameToUnicode** function converts the canonical form of a host name to its Unicode version.

Parameters

lpszHostName

Pointer to the host name as a null-terminated string. This parameter cannot be a NULL pointer or a zero length string.

lpszUnicodeName

Pointer to the string buffer that will contain the original Unicode version of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer.

nMaxLength

The maximum number of characters that can be copied to the *lpszUnicodeName* string buffer. This parameter cannot be zero, and must include the terminating null character.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The **DnsHostNameToUnicode** function will convert the encoded ASCII version of a host name to its Unicode version. Although any valid host name is accepted by this function, it is intended to convert a Punycode encoded host name to its original Unicode character encoding.

If the Unicode version of this function is used, the value returned in *lpszUnicodeName* will be a Unicode string using UTF-16 encoding. If the ANSI version of this function, the value returned will be a Unicode string using UTF-8 encoding. To display a UTF-8 encoded host name, your application will need to convert it to UTF-16 using the **MultiByteToWideChar** function.

Although this function performs checks to ensure that the *lpszHostName* parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the existence of the domain name. To check if the host name exists and has a valid IP address, use the [DnsGetHostAddress](#) function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csdns10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

DnsInitialize Function

```
BOOL WINAPI DnsInitialize(  
    LPCTSTR lpszLicenseKey,  
    LPINITDATA lpData  
);
```

The **DnsInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpData

Pointer to an INITDATA data structure. This parameter may be NULL if the initialization data for the library is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **DnsGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must insure that this function is called with a valid license key.

If the *lpData* argument is specified, it must point to an INITDATA structure which has been initialized by setting the *dwSize* member to the size of the structure. All other structure members should be set to zero. If the function is successful, then the INITDATA structure will be filled with identifying information about the library.

Although it is only required that **DnsInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **DnsUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **DnsInitialize** or **DnsUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

DnsMatchHostName Function

```
BOOL WINAPI DnsMatchHostName(  
    HCLIENT hClient,  
    LPCTSTR lpszHostName,  
    LPCTSTR lpszHostMask  
    BOOL bResolve  
);
```

The **DnsMatchHostName** function matches a host name against one or more strings that may contain wildcards.

Parameters

hClient

Handle to the client session.

lpszHostName

A pointer to a string which specifies the host name or IP address to match.

lpszHostMask

A pointer to a string which specifies one or more values to match against the host name. The asterisk character can be used to match any number of characters in the host name, and the question mark can be used to match any single character. Multiple values may be specified by separating them with a semicolon.

bResolve

A boolean value which specifies if the host name or IP address should be resolved when matching the host against the mask string. If this parameter is non-zero, two checks against the host mask string will be performed; once for the host name specified and once for its IP address. If this parameter is zero, then the match is made only against the host name string provided.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **DnsGetLastError**.

Remarks

The **DnsMatchHostName** function provides a convenient way for an application to determine if a given host name matches one or more mask strings which may contain wildcard characters. For example, the host name could be "www.microsoft.com" and the host mask string could be "*.microsoft.com". In this example, the function would return a non-zero value indicating the host name matched the mask. However, if the mask string was "*.net" then the function would return zero, indicating that there was no match. Multiple mask values can be combined by separating them with a semicolon; for example, the mask "*.com;*.org" would match any host name in either the .com or .org top-level domains.

If an internationalized domain name (IDN) is specified, it will be converted internally to an ASCII string using Punycode encoding. The host mask will be matched against this encoded version of the host name, not its Unicode version.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetAddress](#), [DnsGetHostAddress](#), [DnsGetHostName](#), [DnsGetLocalAddress](#)

DnsNormalizeHostName Function

```
INT WINAPI DnsNormalizeHostName(  
    LPCTSTR LpszHostName,  
    LPTSTR LpszNormalized,  
    INT nMaxLength  
);
```

The **DnsNormalizeHostName** function returns the canonical form of a host name in the specified buffer.

Parameters

LpszHostName

Pointer to the host name as a null-terminated string. This parameter cannot be a NULL pointer or a zero length string.

LpszNormalized

Pointer to the string buffer that will contain the canonical form of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer.

nMaxLength

The maximum number of characters that can be copied to the *LpszNormalized* string buffer. This parameter cannot be zero, and must include the terminating null character.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The **DnsNormalizeHostName** function will remove all leading and trailing whitespace characters from the host name and fold all upper-case characters to lower-case. If an internationalized domain name (IDN) containing Unicode characters is passed to this function, it will be converted to an ASCII compatible format for domain names.

If the Unicode version of this function is used, the host name will be converted from UTF-16 to UTF-8 and then processed. If you are unsure if an internationalized domain name will be specified as the host name, it is recommended that you use the Unicode version.

Although this function performs checks to ensure that the *LpszHostName* parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the existence of the domain name. To check if the host name exists and has a valid IP address, use the [DnsGetHostAddress](#) function.

It is recommended that you use this function if your application needs to store the host name, and if accepts a host name as user input. It is not necessary to call this function prior to calling the other DNS API functions that accept a host name as a parameter. Those functions already normalize the host name and perform checks to ensure it is in the correct format.

If the *LpszHostName* parameter specifies a valid IPv4 or IPv6 address string instead of a host name, this function will return a copy of that IP address in the buffer provided by the caller. This allows the function to be used in cases where a user may input either a host name or IP address. To determine if the IP address has a corresponding host name, use the [DnsGetHostName](#) method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetHostAddress](#), [DnsGetHostName](#), [DnsHostNameToUnicode](#)

DnsRegisterServer Function

```
INT WINAPI DnsRegisterServer(  
    HCLIENT hClient,  
    INT nServer,  
    LPCTSTR lpszHostAddress,  
    INT nPort  
);
```

The **DnsRegisterServer** function registers a nameserver with the current client session. The nameserver is used to resolve queries issued by the client, such as returning the IP address for a given host name. At least one nameserver must be registered by the client before queries are issued.

Parameters

hClient

Handle to the client session.

nServer

The index into the client nameserver table. This index, starting at 0, is used to specify which slot in the client's nameserver table will be used to store the nameserver information.

lpszHostAddress

A pointer to a string that specifies the IP address of the nameserver to be registered. Note that hostnames cannot be specified.

nPort

The port number that the specified nameserver is accepting queries on. This value may be set to zero, in which case it will use the default port value.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsCreateHandle](#), [DnsGetServerAddress](#), [DnsGetServerPort](#), [DnsUnregisterServer](#)

DnsReset Function

```
INT WINAPI DnsReset(  
    HCLIENT hClient  
);
```

The **DnsReset** function resets the current state of the client session. The timeout and retry counts are set to their default values, the local domain name is cleared and all registered servers are removed from the client nameserver table.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsRegisterServer](#), [DnsSetLocalDomain](#), [DnsSetRetryCount](#), [DnsSetTimeout](#), [DnsUnregisterServer](#)

DnsSetHostFile Function

```
INT WINAPI DnsSetHostFile(  
    HCLIENT hClient,  
    LPCTSTR lpszFileName  
);
```

The **DnsSetHostFile** function specifies the name of an alternate file to use when resolving hostnames and IP addresses. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **DnsGetHostAddress** and **DnsGetHostName** functions. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

```
ipaddress hostname [hostalias ...]
```

For example, one typical entry maps the name "localhost" to the local loopback IP address. This would be entered as:

```
127.0.0.1 localhost
```

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

Parameters

hClient

Handle to the client session.

lpszFileName

Pointer to a string that specifies the name of the file. If the parameter is NULL, then the current host file is cleared from the cache and only the default host file will be used to resolve hostnames and addresses.

Return Value

If the function succeeds, the return value is the number of entries in the host file. A return value of DNS_ERROR indicates failure. To get extended error information, call **DnsGetLastError**.

Remarks

This function loads the file into memory allocated for the current thread. If the contents of the file have changed after the function has been called, those changes will not be reflected when resolving hostnames or addresses. To reload the host file from disk, call this function again with the same file name. To remove the alternate host file from memory, specify a NULL pointer as the parameter.

If a host file has been specified, it is processed before the default host file when resolving a hostname into an IP address, or an IP address into a hostname. If the host name or address is not found, or no host file has been specified, a nameserver lookup is performed.

To determine if an alternate host file has been specified, use the **DnsGetHostFile** function. A return value of zero indicates that no alternate host file has been cached for the current thread.

A system may have a default host file, which is used to resolve hostnames before performing a nameserver lookup. To determine the name of this file, use the **DnsGetDefaultHostFile** function. It is not necessary to specify this default host file, since it is always used to resolve host names and addresses.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetDefaultHostFile](#), [DnsGetHostAddress](#), [DnsGetHostFile](#), [DnsGetHostName](#)

DnsSetLastError Function

```
VOID WINAPI DnsSetLastError(  
    DWORD dwErrorCode  
);
```

The **DnsSetLastError** function sets the error code for the current thread. This function is typically used to clear the last error by passing a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the error code for the current thread. A value of zero clears the last error code.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or DNS_ERROR. Those functions which call **DnsSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **DnsGetLastError** function. The use of **DnsGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

See Also

[DnsGetErrorString](#), [DnsGetLastError](#)

DnsSetLocalDomain Function

```
INT WINAPI DnsSetLocalDomain(  
    HCLIENT hClient,  
    LPCTSTR lpszDomain  
);
```

Parameters

hClient

Handle to the client session.

lpszDomain

Pointer to the string which contains the local domain name. This is used as a default value when a query does not explicitly specify a domain name.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetLocalDomain](#), [DnsGetRetryCount](#), [DnsGetTimeout](#), [DnsReset](#), [DnsSetRetryCount](#), [DnsSetTimeout](#)

DnsSetResolverOptions Function

```
DWORD WINAPI DnsSetResolverOptions(  
    HCLIENT hClient,  
    DWORD dwOptions  
);
```

The **DnsSetResolverOptions** function changes the resolver options for the specified client session.

Parameters

hClient

Handle to the client session.

dwOptions

An unsigned integer that specifies one or more resolver options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
DNS_OPTION_NONE	No additional resolver options specified. This is the default value, and it is recommended that most applications do not specify additional options unless the implications of doing so are understood.
DNS_OPTION_PROMOTE	Promotes the server that successfully completed the last query to the first server that will be used to resolve subsequent queries. This option can improve performance in some cases where one or more of the registered servers are non-responsive. This option takes precedence over the DNS_OPTION_ROTATE option.
DNS_OPTION_ROTATE	Enables a round-robin selection of nameservers when performing queries. Normally each nameserver is queried in the same order. This option rotates the available nameservers so a different server is used with each query.
DNS_OPTION_AUTHONLY	Require the answer from the nameserver to be authoritative, not from the server's cache. This option is included for future expansion as most servers do not support this feature and will ignore it.
DNS_OPTION_PRIMARY	Queries are only accepted from the primary nameserver. This option is included for future expansion as most servers do not support this feature and will ignore it.
DNS_OPTION_NORECURSE	Disable the sending of recursive queries to the nameserver. Specifying this option will disable the bit in the DNS request header that specifies recursion is desired.
DNS_OPTION_NOSEARCH	Disable additional queries of higher domains in the

	search list if the host name cannot be resolved. If this option is specified, and the host name cannot be resolved using the local domain name an error is returned immediately. This option is ignored if no local domain has been specified or if the DNS_OPTION_NOSUFFIX option has been specified.
DNS_OPTION_NOSUFFIX	Disable additional queries using the local domain name if the host name is not a fully qualified domain name and cannot be resolved. This option is ignored if no local domain has been specified.
DNS_OPTION_NONAMECHECK	Disable checking the host name for invalid characters, such as the underscore and control characters. By default, host names are checked to ensure they're valid before submitting a query to the nameserver.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **DnsGetLastError**.

Remarks

The **DnsSetResolverOptions** function changes the resolver options for the specified client session, modifying how nameserver queries are processed. It is recommended that most applications do not specify any resolver options and use the default behavior. Specifying these options without understanding how they can affect standard queries can result in unexpected failures. In particular, caution should be used when specifying the DNS_OPTION_NORECURSE and DNS_OPTION_NOSEARCH options as they change the normal process of resolving a host name.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

See Also

[DnsCreateHandle](#), [DnsGetResolverOptions](#)

DnsSetRetryCount Function

```
INT WINAPI DnsSetRetryCount(  
    HCLIENT hClient,  
    INT nRetries  
);
```

The **DnsSetRetryCount** function sets the number of attempts that the client will make attempting to resolve a query. When used in conjunction with the **DnsSetTimeout** function, it determines the total amount of time the client will spend attempting to resolve a query.

Parameters

hClient

Handle to the client session.

nRetries

The number of attempts the client will make, per nameserver, to resolve a query.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The retry count determines the amount of time the client will wait for a response from each query, the effective amount of time the client will wait increases with each nameserver and the total number of retries specified. For example, two nameservers registered with the client, with a default of 4 retries per nameserver and a timeout value of 10 seconds, would cause the client to wait a total of 80 seconds until it returns an error indicating that it was unable to resolve the query.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetRetryCount](#), [DnsGetTimeout](#), [DnsSetTimeout](#)

DnsSetTimeout Function

```
INT WINAPI DnsSetTimeout(  
    HCLIENT hClient,  
    INT nTimeout  
);
```

The **DnsSetTimeout** function sets the number of seconds that the client will wait for a response from a nameserver. The timeout value is used each time a server in the client's nameserver table is queried. When used in conjunction with the **DnsSetRetryCount** function, it determines the total amount of time the client will spend attempting to resolve a query.

Parameters

hClient

Handle to the client session.

nTimeout

The number of seconds until the client times out waiting for a response from a nameserver.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The timeout value determines the amount of time the client will wait for a response from each query, the effective amount of time the client will wait increases with each nameserver and the total number of retries specified. For example, with two nameservers registered with the client, with a default of 4 retries per nameserver and a timeout value of 10 seconds, would cause the client to wait a total of 80 seconds until it returns an error indicating that it was unable to resolve the query.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

See Also

[DnsGetRetryCount](#), [DnsGetTimeout](#), [DnsSetRetryCount](#)

DnsUninitialize Function

```
VOID WINAPI DnsUninitialize();
```

The **DnsUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **DnsInitialize** call before it can call any of the other the library functions. When it has completed the use of library, the application must call **DnsUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There must be a call to **DnsUninitialize** for every successful call to **DnsInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsInitialize](#), [DnsCloseHandle](#), [DnsCreateHandle](#)

DnsUnregisterServer Function

```
INT WINAPI DnsUnregisterServer(  
    HCLIENT hClient,  
    INT nServer  
);
```

The **DnsUnregisterServer** function removes the specified nameserver information from the client. Unregistering a server prevents the client from using that server to satisfy subsequent DNS queries.

Parameters

hClient

Handle to the client session.

nServer

The index into the client's nameserver table. This index is the same value that is passed to the **DnsRegisterServer** function when the nameserver is registered.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is DNS_ERROR. To get extended error information, call **DnsGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csdnsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DnsGetServerAddress](#), [DnsGetServerPort](#), [DnsRegisterServer](#), [DnsReset](#)

Domain Name Service Data Structures

- HOSTENT
- INITDATA
- INTERNET_ADDRESS

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

HOSTENT Structure

This structure is used by the [DnsGetHostByAddress](#) and [DnsGetHostByName](#) functions to return information about a specified host. The application must never attempt to modify this structure or to free any of its components.

Only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other function calls. This is the same data structure used by the Windows Sockets API.

```
typedef struct _HOSTENT
{
    char *   h_name;
    char **  h_aliases;
    short    h_addrtype;
    short    h_length;
    char **  h_addr_list;
} HOSTENT, *LPHOSTENT;
```

Members

h_name

The fully qualified domain name (FQDN) that caused the nameserver server to return a reply.

h_aliases

A NULL-terminated array of alternate names.

h_addrtype

The type of address being returned.

h_length

The length, in bytes, of each address.

h_addr_list

A NULL-terminated list of addresses for the host. Addresses are returned in network byte order. The macro **h_addr** is defined to be **h_addr_list[0]** for compatibility with older software.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include winsock.h.

INITDATA Structure

This structure contains information about the SocketTools library when the initialization function returns.

```
typedef struct _INITDATA
{
    DWORD        dwSize;
    DWORD        dwVersionMajor;
    DWORD        dwVersionMinor;
    DWORD        dwVersionBuild;
    DWORD        dwOptions;
    DWORD_PTR    dwReserved1;
    DWORD_PTR    dwReserved2;
    TCHAR        szDescription[128];
} INITDATA, *LPINITDATA;
```

Members

dwSize

Size of this structure. This structure member must be set to the size of the structure prior to calling the initialization function. Failure to do so will cause the function to fail.

dwVersionMajor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the major version number for the library.

dwVersionMinor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the minor version number for the library.

dwVersionBuild

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the build number for the library.

dwOptions

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved1

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved2

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

szDescription

A null terminated string which describes the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

INTERNET_ADDRESS Structure

This structure represents a numeric IPv4 or IPv6 address in network byte order.

```
typedef struct _INTERNET_ADDRESS
{
    INT    ipFamily;
    BYTE   ipNumber[16];
} INTERNET_ADDRESS, *LPINTERNET_ADDRESS;
```

Members

ipFamily

An integer which identifies the type of IP address. It will be one of the following values:

Constant	Description
INET_ADDRESS_UNKNOWN	The address has not been specified or the bytes in the <i>ipNumber</i> array does not represent a valid address. Functions which populate this structure will use this value to indicate that the address cannot be determined.
INET_ADDRESS_IPV4	Specifies that the address is in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address is in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

ipNumber

A byte array which contains the numeric form of the IP address. This array is large enough to store both IPv4 (32 bit) and IPv6 (128 bit) addresses. The values are stored in network byte order.

Remarks

The **INTERNET_ADDRESS** structure is used by some functions to represent an Internet address in a binary format that is compatible with both IPv4 and IPv6 addresses. Applications that use this structure should consider it to be opaque, and should not modify the contents of the structure directly.

For compatibility with legacy applications that expect an IP address to be 32 bits and stored in an unsigned integer, you can copy the first four bytes of the *ipNumber* array using the **CopyMemory** function or equivalent. Note that if this is done, your application should always check the *ipFamily* member first to make sure that it is actually an IPv4 address.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Domain Name Service Constants

Value	Constant	Description
0	DNS_RECORD_NONE	No record type
1	DNS_RECORD_ADDRESS	Host address
2	DNS_RECORD_NS	Authoritative nameserver
5	DNS_RECORD_CNAME	Canonical name (alias)
6	DNS_RECORD_SOA	Start of Authority
11	DNS_RECORD_WKS	Well known services
12	DNS_RECORD_PTR	Domain name
13	DNS_RECORD_HINFO	Host information
14	DNS_RECORD_MINFO	Mailbox information
15	DNS_RECORD_MX	Mail exchange host
16	DNS_RECORD_TXT	Text strings
29	DNS_RECORD_LOC	Location information
100	DNS_RECORD_UINFO	User information
101	DNS_RECORD_UID	User ID
102	DNS_RECORD_GID	Group ID

Encoding and Compression Library

Encode and decode files using standard algorithms such as base64, uuencode and quoted-printable. The library can also be used to compress and expand files, as well as encrypt or decrypt file data using AES encryption.

Reference

- [Functions](#)

Library Information

File Name	CSNCDV10.DLL
Version	10.0.1468.2518
LibID	15999BB0-0394-4284-9E69-2DC698118E2F
Import Library	CSNCDV10.LIB
Dependencies	None
Standards	RFC 1738, RFC 1951, RFC 2045

Overview

The Encoding and Compression library provides functions for encoding and decoding binary files, typically attachments to email messages. The process of encoding converts the contents of a binary file to printable 7-bit ASCII text. Decoding reverses the process, converting a previously encoded text file back into a binary file.

There are two primary types of encoding methods used with various Internet applications: base64 and uuencode. The base64 algorithm is most commonly used with email attachments, and is often referred to as MIME encoding since this is the encoding method specified in the MIME standards document. The uuencode algorithm (so called because the programs to perform the encoding were called uuencode and uudecode) is often used when attaching binary files to Usenet newsgroup posts. The library also supports an alternate encoding format called yEnc which is also widely used to attach files to Usenet posts.

In addition to encoding and decoding data files, this library includes functions to compress and expand data, as well as encrypt and decrypt files using 256-bit AES encryption.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Encoding and Compression Functions

Function	Description
AesDecryptBuffer	Decrypt a buffer previously encrypted using the AesEncryptBuffer function
AesDecryptFile	Decrypt a file previously encrypted using the <code>strong>AesEncryptFile</code> function
AesDecryptString	Decrypt a string previously encrypted using the AesEncryptString function
AesEncryptBuffer	Encrypt a memory buffer using 256-bit AES encryption
AesEncryptFile	Encrypt the contents of a file using 256-bit AES encryption
AesEncryptString	Encrypt a null-terminated string using 256-bit AES encryption
CompressBuffer	Compress the contents of the specified buffer
CompressFile	Compress the contents of the specified file
CompressFileEx	Compress the contents of the specified file with additional options
DecodeBuffer	Decode an encoded string, storing the result in the specified buffer
DecodeFile	Decode the contents of a file using the specified decoding method
EncodeBuffer	Encode the contents of a buffer using the specified encoding method
EncodeFile	Encode the contents of a file using the specified encoding method
ExpandBuffer	Expand the contents of a previously compressed buffer
ExpandFile	Expand the contents of a previously compressed file
ExpandFileEx	Expand the contents of a previously compressed file with additional options
GetMessageDigest	Compute a message digest for a buffer, string or data file
IsUnicodeText	Determine if a text buffer contains valid Unicode characters
UnicodeDecodeText	Decode UTF-8 encoded text and return a localized string
UnicodeEncodeText	Encode a text buffer and return a UTF-8 encoded string

AesDecryptBuffer Function

```
BOOL WINAPI AesDecryptBuffer(  
    LPCTSTR lpszPassword,  
    LPVOID lpvInputBuffer,  
    DWORD dwInputSize  
    LPVOID lpvOutputBuffer,  
    LPDWORD lpdwOutputSize  
);
```

The **AesDecryptBuffer** function decrypts the contents of a memory buffer.

Parameters

lpszPassword

A pointer to a null terminated string of characters that will be used to generate the decryption key. This parameter may be NULL or a zero-length string, in which case a default internal hash value is used to decrypt the data. Password strings that exceed 215 characters will be truncated.

lpvInputBuffer

A pointer to the buffer which contains the data to be decrypted.

dwInputSize

The number of bytes in the input buffer. This value must be greater than zero.

lpvOutputBuffer

A pointer to the buffer which will contain the decrypted data when the function returns. This memory must be allocated by the application and be large enough to contain all of the decrypted data. If the output buffer is not large enough, the function will fail. This parameter cannot be NULL.

lpdwOutputSize

A pointer to the number of bytes that may be copied into the output buffer. This parameter must be initialized to a non-zero value. When the function returns, the actual number of bytes of decrypted data is returned in this parameter. This parameter cannot be NULL.

Return Value

A non-zero value is returned if the data was successfully decrypted. A zero value indicates that the data could not be decrypted. To get extended error information, call **GetLastError**.

Remarks

The **AesDecryptBuffer** function will decrypt a block of memory using a 256-bit AES (Advanced Encryption Standard) algorithm and returns a copy of the decrypted data to the caller. The password (or passphrase) provided by the caller is used to generate a SHA-256 hash value which is used as part of the decryption process. The *lpszPassword* value must be identical to the value used to encrypt the data using the **AesEncryptBuffer** function.

Due to how the SHA-256 hash is generated, this function cannot be used to decrypt data that was encrypted using another third-party library. It can only be used to decrypt data that was previously encrypted using **AesEncryptBuffer**.

If you wish to decrypt the contents of a file, use the **AesDecryptFile** function.

This function uses the Microsoft CryptoAPI and the RSA AES cryptographic provider. This provider may not be available in some languages, countries or regions. The availability of this provider may also be constrained by cryptography export restrictions imposed by the United States or other

countries. If the required cryptographic provider is not available, the function will fail.

Example

```
BOOL bIsDecrypted = FALSE;
DWORD dwOutputSize = MAX_BUFFER_SIZE;
BYTE *lpOutputBuffer = (BYTE *)LocalAlloc(LPTR, dwOutputSize);

if (lpOutputBuffer != NULL)
{
    bIsDecrypted = AesDecryptBuffer(lpszPassword,
                                   lpInputBuffer,
                                   dwInputSize,
                                   lpOutputBuffer,
                                   &dwOutputSize);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csncdv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AesDecryptFile](#), [AesDecryptString](#), [AesEncryptBuffer](#), [CompressBuffer](#)

AesDecryptFile Function

```
BOOL WINAPI AesDecryptFile(  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszInputFile,  
    LPCTSTR lpszOutputFile  
);
```

The **AesDecryptFile** function decrypts the contents of a file.

Parameters

lpszPassword

A pointer to a null terminated string of characters that will be used to generate the decryption key. This parameter may be NULL or a zero-length string, in which case a default internal hash value is used to decrypt the data. Password strings that exceed 215 characters will be truncated.

lpszInputFile

A pointer to a null terminated string which specifies the name of the file to be decrypted. The file must exist, and it must be a regular file that can be opened for reading by the current process. An error will be returned if a character device, such as CON: is specified as the file name.

lpszOutputFile

A pointer to a null terminated string which specifies the name of the file that will contain the decrypted data. If the file exists, it will be overwritten. It must be a regular file that can be opened for writing by the current process. An error will be returned if a character device, such as CON: is specified as the file name.

Return Value

A non-zero value is returned if the string was successfully encrypted. A zero value indicates that the string could not be encrypted. To get extended error information, call **GetLastError**.

Remarks

The **AesDecryptFile** function will decrypt the contents of a file using a 256-bit AES (Advanced Encryption Standard) algorithm and stores the decrypted data in the specified output file. The password (or passphrase) provided by the caller is used to generate a SHA-256 hash value which is used as part of the decryption process. The *lpszPassword* value must be identical to the value used to encrypt the data using the **AesEncryptFile** function.

Due to how the SHA-256 hash is generated, this function cannot be used to decrypt files that were encrypted using another third-party library. It can only be used to decrypt data that was previously encrypted using **AesEncryptFile**.

A temporary file is created during the decryption process and the output file is created or overwritten only if the input file could be successfully decrypted. If the decryption fails, no output file will be created.

If you wish to decrypt the contents of a memory buffer or string, use the **AesDecryptBuffer** or **AesDecryptString** functions.

This function uses the Microsoft CryptoAPI and the RSA AES cryptographic provider. This provider may not be available in some languages, countries or regions. The availability of this provider may also be constrained by cryptography export restrictions imposed by the United States or other countries. If the required cryptographic provider is not available, the function will fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csncdv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AesDecryptBuffer](#), [AesDecryptString](#), [AesEncryptFile](#)

AesDecryptString Function

```
BOOL WINAPI AesDecryptString(  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszInputString,  
    LPCTSTR lpszOutputString,  
    LONG nMaxLength  
);
```

The **AesDecryptString** function decrypts the contents of a string.

Parameters

lpszPassword

A pointer to a null terminated string of characters that will be used to generate the decryption key. This parameter may be NULL or a zero-length string, in which case a default internal hash value is used to decrypt the data. Password strings that exceed 215 characters will be truncated.

lpszInputString

A pointer to a null terminated string which contains the data to be decrypted. The encrypted input data must be base64 encoded and identical to the encrypted string returned by the **AesEncryptString** function. If this parameter is NULL or points to an empty string the function will fail.

lpszOutputString

A pointer to the buffer which will contain the decrypted string data. This parameter cannot be NULL.

nMaxLength

The maximum number of characters that can be copied to the output string buffer. The output buffer must be large enough to store the complete decrypted string and is terminated with a null character. This value must be greater than zero. If the output string buffer is not large enough, the function will fail.

Return Value

A non-zero value is returned if the string was successfully encrypted. A zero value indicates that the string could not be decrypted. To get extended error information, call **GetLastError**.

Remarks

The **AesDecryptString** function will decrypt a string using a 256-bit AES (Advanced Encryption Standard) algorithm and returns a copy of the decrypted string to the caller. The password (or passphrase) provided by the caller is used to generate a SHA-256 hash value which is used as part of the decryption process. The *lpszPassword* value must be identical to the value used to encrypt the data using the **AesEncryptString** function.

Due to how the SHA-256 hash is generated, this function cannot be used to decrypt strings that were encrypted using another third-party library. It can only be used to decrypt strings that were previously encrypted using **AesEncryptString**.

If you wish to decrypt the contents of a file, use the **AesDecryptFile** function.

This function uses the Microsoft CryptoAPI and the RSA AES cryptographic provider. This provider may not be available in some languages, countries or regions. The availability of this provider may also be constrained by cryptography export restrictions imposed by the United States or other countries. If the required cryptographic provider is not available, the function will fail.

Example

```
BOOL bIsDecrypted = FALSE;
LPCTSTR lpszPassword = _T("NFr-E{Ki3_1w0iV+LI@z}");
TCHAR szDecryptedText[MAX_STRING_LENGTH];

bIsDecrypted = AesDecryptString(lpszPassword,
                               szEncryptedText,
                               szDecryptedText,
                               MAX_STRING_LENGTH);

if (bIsDecrypted)
{
    _tprintf(_T("The decrypted string is \"%s\"\n"), szDecryptedText);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csncdv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AesDecryptBuffer](#), [AesDecryptFile](#), [AesEncryptString](#)

AesEncryptBuffer Function

```
BOOL WINAPI AesEncryptBuffer(  
    LPCTSTR lpszPassword,  
    LPVOID lpvInputBuffer,  
    DWORD dwInputSize  
    LPVOID lpvOutputBuffer,  
    LPDWORD lpdwOutputSize  
);
```

The **AesEncryptBuffer** function encrypts the contents of a memory buffer.

Parameters

lpszPassword

A pointer to a null terminated string of characters that will be used to generate the encryption key. Passwords are case sensitive and must match exactly when decrypting data that was previously encrypted using this function. This parameter may be NULL or a zero-length string, in which case a default internal hash value is used. Password strings that exceed 215 characters will be truncated.

lpvInputBuffer

A pointer to the buffer which contains the data to be encrypted.

dwInputSize

The number of bytes in the input buffer. This value must be greater than zero.

lpvOutputBuffer

A pointer to the buffer which will contain the encrypted data when the function returns. This memory must be allocated by the application and be large enough to contain all of the encrypted data. The amount encrypted data returned will always be larger than the amount of data specified by *dwInputSize*. If the output buffer is not large enough, the function will fail. This parameter cannot be NULL.

lpdwOutputSize

A pointer to the number of bytes that may be copied into the output buffer. This parameter must be initialized to a non-zero value. When the function returns, the actual number of bytes of encrypted data is returned in this parameter. This parameter cannot be NULL.

Return Value

A non-zero value is returned if the data was successfully encrypted. A zero value indicates that the data could not be encrypted. To get extended error information, call **GetLastError**.

Remarks

The **AesEncryptBuffer** function will encrypt a block of memory using a 256-bit AES (Advanced Encryption Standard) algorithm and returns a copy of the encrypted data to the caller. The password (or passphrase) provided by the caller is used to generate a SHA-256 hash value which is used as part of the encryption process. The identical password is required to decrypt the data using the **AesDecryptBuffer** function.

It is recommended that most applications specify a password value. If the *lpszPassword* parameter is NULL or specifies a zero-length string, a default internal hash value is used. This means that any other application which uses a NULL password value will be able to decrypt the data. If the Unicode version of this function is called, the *lpszPassword* value will be encoded using UTF-8

prior to the hash value being generated.

Due to how the SHA-256 hash is generated, the encrypted data cannot be decrypted using another third-party library with the same password value. It can only be decrypted using the **AesDecryptBuffer** function.

The amount of encrypted data returned by this function will always be somewhat larger than original unencrypted data. If your application dynamically allocates a block of memory to store the encrypted data, provide a maximum buffer size that is at least several hundred bytes larger than the unencrypted data. If the output buffer provided is not large enough, the function will fail and the **GetLastError** function will return ERROR_INSUFFICIENT_BUFFER.

The encrypted data returned by this function can contain embedded nulls and should be typically handled as unsigned char (byte) values. If you wish to encrypt strings and store the encrypted values, use the **AesEncryptString** function. It will perform the same 256-bit AES encryption, but return the encrypted data as a base64 encoded string rather than binary data.

If your application is also using the **CompressBuffer** function to compress the data, it is recommended that you call **CompressBuffer** before calling **AesEncryptBuffer**. You will typically achieve a better compression rate on unencrypted data than attempting to compress data which has been encrypted with this function.

If you wish to encrypt the contents of a file, use the **AesEncryptFile** function.

This function uses the Microsoft CryptoAPI and the RSA AES cryptographic provider. This provider may not be available in some languages, countries or regions. The availability of this provider may also be constrained by cryptography export restrictions imposed by the United States or other countries. If the required cryptographic provider is not available, the function will fail.

Example

```
BOOL bIsEncrypted = FALSE;
DWORD dwOutputSize = MAX_BUFFER_SIZE;
BYTE *lpOutputBuffer = (BYTE *)LocalAlloc(LPTR, dwOutputSize);

if (lpOutputBuffer != NULL)
{
    bIsEncrypted = AesEncryptBuffer(lpszPassword,
                                   lpInputBuffer,
                                   dwInputSize,
                                   lpOutputBuffer,
                                   &dwOutputSize);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csncdv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AesDecryptBuffer](#), [AesEncryptFile](#), [AesEncryptString](#), [CompressBuffer](#)

AesEncryptFile Function

```
BOOL WINAPI AesEncryptFile(  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszInputFile,  
    LPCTSTR lpszOutputFile  
);
```

The **AesEncryptFile** function encrypts the contents of a file.

Parameters

lpszPassword

A pointer to a null terminated string of characters that will be used to generate the encryption key. Passwords are case sensitive and must match exactly when decrypting data that was previously encrypted using this function. This parameter may be NULL or a zero-length string, in which case a default internal hash value is used. Password strings that exceed 215 characters will be truncated.

lpszInputFile

A pointer to a null terminated string which specifies the name of the file to be encrypted. The file must exist, and it must be a regular file that can be opened for reading by the current process. An error will be returned if a character device, such as CON: is specified as the file name.

lpszOutputFile

A pointer to a null terminated string which specifies the name of the file that will contain the encrypted data. If the file exists, it will be overwritten. It must be a regular file that can be opened for writing by the current process. An error will be returned if a character device, such as CON: is specified as the file name.

Return Value

A non-zero value is returned if the string was successfully encrypted. A zero value indicates that the string could not be encrypted. To get extended error information, call **GetLastError**.

Remarks

The **AesEncryptFile** function will encrypt the contents of a file using a 256-bit AES (Advanced Encryption Standard) algorithm and stores the encrypted data in the specified output file. The password (or passphrase) provided by the caller is used to generate a SHA-256 hash value which is used as part of the encryption process. The identical password is required to decrypt the data using the **AesDecryptFile** function.

It is recommended that most applications specify a password value. If the *lpszPassword* parameter is NULL or specifies a zero-length string, a default internal hash value is used. This means that any other application which uses a NULL password value will be able to decrypt the data. If the Unicode version of this function is called, the *lpszPassword* value will be encoded using UTF-8 prior to the hash value being generated.

Due to how the SHA-256 hash is generated, the encrypted data cannot be decrypted using another third-party library with the same password value. It can only be decrypted using the **AesDecryptFile** function.

A temporary file is created during the encryption process and the output file is created or overwritten only if the input file could be successfully encrypted. If the encryption fails, no output

file will be created.

The input file contents will always be processed as a binary data stream. If you use this function to encrypt a text file, the output file will contain binary characters, not printable text. If you wish to transfer or store the encrypted data as text, it should be encoded using the **EncodeFile** function after calling **AesEncryptFile**.

If your application is also using the **CompressFile** function to compress the data, it is recommended that you call **CompressFile** before calling **AesEncryptFile**. You will typically achieve a better compression rate on unencrypted data than attempting to compress data which has been encrypted with this function.

If you wish to encrypt the contents of a memory buffer or string, use the **AesEncryptBuffer** or **AesEncryptString** functions.

This function uses the Microsoft CryptoAPI and the RSA AES cryptographic provider. This provider may not be available in some languages, countries or regions. The availability of this provider may also be constrained by cryptography export restrictions imposed by the United States or other countries. If the required cryptographic provider is not available, the function will fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csncdv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AesDecryptFile](#), [AesEncryptBuffer](#), [AesEncryptString](#), [CompressFile](#), [EncodeFile](#)

AesEncryptString Function

```
BOOL WINAPI AesEncryptString(  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszInputString,  
    LONG nInputLength  
    LPCTSTR lpszOutputString,  
    LONG nMaxLength  
);
```

The **AesEncryptString** function encrypts the contents of a string.

Parameters

lpszPassword

A pointer to a null terminated string of characters that will be used to generate the encryption key. Passwords are case sensitive and must match exactly when decrypting data that was previously encrypted using this function. This parameter may be NULL or a zero-length string, in which case a default internal hash value is used. Password strings that exceed 215 characters will be truncated.

lpszInputString

A pointer to a null terminated string which contains the text to be encrypted. If the Unicode version of this function is called, the string is automatically UTF-8 encoded prior to being encrypted.

nInputLength

The number of characters in the input string which will be encrypted. If this value is -1, the length of the input string is determined automatically by counting the number of characters up to the terminating null byte. Note that with Unicode strings, this value must represent the number of characters in the string, not the number of bytes allocated for the string.

lpszOutputString

A pointer to the buffer which will contain a base64 encoded encrypted string when the function returns. This memory must be allocated by the application and be large enough to contain all of the encrypted data. The amount encrypted data returned will always be larger than the amount of data specified by *nInputLength*. If the output string buffer is not large enough, the function will fail. This parameter cannot be NULL.

nMaxLength

The maximum number of characters that can be copied to the output string buffer. The output buffer must be large enough to store the complete encrypted string which is encoded using base64 and terminated with a null character. This value must be greater than zero. If the output string buffer is not large enough, the function will fail.

Return Value

A non-zero value is returned if the string was successfully encrypted. A zero value indicates that the string could not be encrypted. To get extended error information, call **GetLastError**.

Remarks

The **AesEncryptString** function will encrypt a string using a 256-bit AES (Advanced Encryption Standard) algorithm and returns a copy of the encrypted data as a base64 encoded string to the caller. The password (or passphrase) provided by the caller is used to generate a SHA-256 hash value which is used as part of the encryption process. The identical password is required to

decrypt the data using the **AesDecryptString** function.

It is recommended that most applications specify a password value. If the *lpzPassword* parameter is NULL or specifies a zero-length string, a default internal hash value is used. This means that any other application which uses a NULL password value will be able to decrypt the data. If the Unicode version of this function is called, the *lpzPassword* value will be encoded using UTF-8 prior to the hash value being generated.

Due to how the SHA-256 hash is generated, the encrypted data cannot be decrypted using another third-party library with the same password value. It can only be decrypted using the **AesDecryptString** function.

The amount of encrypted data returned by this function will always be larger than original unencrypted data. If your application dynamically allocates a block of memory to store the encrypted string, provide a maximum output string size that is at least several hundred bytes larger than the unencrypted data. If the output string is not large enough, the function will fail and the **GetLastError** function will return ERROR_INSUFFICIENT_BUFFER.

The string provided to this function cannot contain embedded nulls and should not be used to encrypt binary data. If you wish to encrypt binary data, use the **AesEncryptBuffer** function. It will perform the same 256-bit AES encryption and return the encrypted data into a buffer provided by the caller.

If you wish to encrypt the contents of a file, use the **AesEncryptFile** function.

This function uses the Microsoft CryptoAPI and the RSA AES cryptographic provider. This provider may not be available in some languages, countries or regions. The availability of this provider may also be constrained by cryptography export restrictions imposed by the United States or other countries. If the required cryptographic provider is not available, the function will fail.

Example

```
BOOL bIsEncrypted = FALSE;
LPCTSTR lpzPassword = _T("NFr-E{Ki3_1w0iV+LI@z}");
LPCTSTR lpzPlainText = _T("The quick brown fox jumped over the lazy dog.");
TCHAR szEncryptedText[MAX_STRING_LENGTH];

bIsEncrypted = AesEncryptString(lpzPassword,
                               lpzPlainText,
                               -1,
                               szEncryptedText,
                               MAX_STRING_LENGTH);

if (bIsEncrypted)
{
    _tprintf(_T("The encrypted string is \"%s\"\n"), szEncryptedText);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csncdv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[AesDecryptString](#), [AesEncryptBuffer](#), [AesEncryptFile](#)

CompressBuffer Function

```
BOOL WINAPI CompressBuffer(  
    LPVOID lpvInputBuffer,  
    DWORD cbInputBuffer  
    LPVOID lpvOutputBuffer,  
    LPDWORD lpcbOutputBuffer  
);
```

The **CompressBuffer** function compresses the contents of the specified buffer.

Parameters

lpvInputBuffer

A pointer to the buffer which contains the data to be compressed.

cbInputBuffer

The number of bytes in the input buffer. This value must be greater than zero.

lpvOutputBuffer

A pointer to the buffer which will contain the compressed data. This parameter may be NULL, which can be used to determine the size of the of the resulting compressed data prior to allocating memory for it.

lpcbOutputBuffer

A pointer to the number of bytes that may be copied into the output buffer. If the *lpvOutputBuffer* parameter is not NULL, this must be initialized to a non-zero value. When the function returns, the actual number of bytes of compressed data is returned in this parameter.

Return Value

A non-zero value is returned if the data was successfully compressed. A zero value indicates that the data could not be compressed.

Remarks

The compression ratio achieved by the **CompressBuffer** function depends on the type of data that is being compressed. The compressed data can be expanded to its original contents by calling the **ExpandBuffer** function.

Example

```
cbOutputBuffer = 0;  
bResult = CompressBuffer(lpInputBuffer,  
                        cbInputBuffer,  
                        NULL,  
                        &cbOutputBuffer);  
  
if (bResult)  
{  
    lpOutputBuffer = (LPBYTE)LocalAlloc(LPTR, cbOutputBuffer);  
    bResult = CompressBuffer(lpInputBuffer,  
                            cbInputBuffer,  
                            lpOutputBuffer,  
                            &cbOutputBuffer);  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csncdv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CompressFile](#), [DecodeFile](#), [EncodeFile](#), [ExpandBuffer](#), [ExpandFile](#)

CompressFile Function

```
BOOL WINAPI CompressFile(  
    LPCTSTR lpszInputFile,  
    LPCTSTR lpszOutputFile  
);
```

The **CompressFile** function compresses the contents of the specified file.

Parameters

lpszInputFile

A pointer to a string which specifies the name of the file to be compressed. The file must exist, and it must be a regular file that can be opened for reading by the current process. An error will be returned if a character device, such as CON:, is specified as the file name.

lpszOutputFile

The name of the file that is to contain the compressed file data. If the file exists, it must be a regular file that can be opened for writing by the current process and will be overwritten. If the file does not exist, it will be created. An error will be returned if a character device, such as CON:, is specified as the file name.

Return Value

A non-zero value is returned if the file was successfully compressed. A zero value indicates that the input file could not be read or that the output file could not be created.

Remarks

The compression ratio achieved by the **CompressFile** function depends on the type of file that is being compressed. The compressed file is not stored in an archive format that is recognized by third-party applications such as PKZip or WinZip.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csncdv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CompressFileEx](#), [DecodeFile](#), [EncodeFile](#), [ExpandFile](#), [ExpandFileEx](#)

CompressFileEx Function

```
BOOL WINAPI CompressFileEx(  
    LPCTSTR lpszInputFile,  
    LPCTSTR lpszOutputFile,  
    DWORD dwCompressionType,  
    DWORD dwCompressionLevel,  
    LPVOID lpvHeaderBlock,  
    DWORD dwHeaderLength  
);
```

The **CompressFileEx** function compresses the contents of the specified file.

Parameters

lpszInputFile

A pointer to a string which specifies the name of the file to be compressed. The file must exist, and it must be a regular file that can be opened for reading by the current process. An error will be returned if a character device, such as CON: is specified as the file name.

lpszOutputFile

The name of the file that is to contain the compressed file data. If the file exists, it must be a regular file that can be opened for writing by the current process and will be overwritten. If the file does not exist, it will be created. An error will be returned if a character device, such as CON: is specified as the file name.

dwCompressionType

A numeric value which determines the algorithm that will be used to compress the data. One of the following values may be specified. By default, the Deflate algorithm is used.

Value	Constant	Description
1	COMPRESSION_TYPE_DEFLATE	A compression algorithm that combines LZ77 algorithm for creating common substrings and Huffman coding to process the different frequencies of byte sequences in the data stream. Deflate is widely used by compression software.
2	COMPRESSION_TYPE_BURROWSWHEELER	A compression algorithm that rearranges blocks of data in sorted order and then uses Huffman coding to process different frequencies of data within the block. Burrows-Wheeler compression provides a better compression ratio

	than the Deflate algorithm, however it requires more resources to perform the compression.
--	--

dwCompressionLevel

A numeric value which specifies the compression level to use. A value of zero specifies that the default compression level appropriate for the selected algorithm should be used, balancing resource usage and the compression ratio of the data. A value of 1 specifies that the compression should be performed using minimal memory resources, at the expense of the compression ratio. The maximum value of 9 specifies that the algorithm should use more memory to achieve the maximum compression ratio. It is recommended that most applications use the default value of zero.

lpvHeaderBlock

A pointer to a block of uncompressed data that is written to the beginning of the output file. If this parameter is NULL and the *dwHeaderLength* parameter is not zero, a null buffer of the specified size is written to the file.

dwHeaderLength

The length of the data buffer to be written to beginning of the output file. If this value is zero, the *lpvHeaderBlock* parameter is ignored and no header block is written.

Return Value

A non-zero value is returned if the file was successfully compressed. A zero value indicates that the input file could not be read or that the output file could not be created.

Remarks

The compression ratio achieved by the **CompressFile** function depends on the type of file that is being compressed. The compressed file is not stored in an archive format that is recognized by third-party applications such as PKZip or WinZip.

The *lpvHeaderBlock* and *dwHeaderLength* parameters can be used to write application-defined data to the beginning of the file, commonly called a header block. Typically this is a fixed-length structure which provides information to the application about the compressed file.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csncdv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CompressBuffer](#), [CompressFile](#), [DecodeFile](#), [EncodeFile](#), [ExpandBuffer](#), [ExpandFile](#), [ExpandFileEx](#)

DecodeBuffer Function

```
LONG WINAPI DecodeBuffer(  
    LPCTSTR lpszInput,  
    LPBYTE lpOutput,  
    LONG cbOutput,  
    DWORD dwOptions  
);
```

The **DecodeBuffer** function decodes an encoded string, and stores the result in the specified buffer.

Parameters

lpszInput

A pointer to a string which contains the base64 encoded text.

lpOutput

A pointer to a byte array buffer which is used to store the decoded data. It is recommended that the buffer be as large as the length of the encoded string.

cbOutput

A long integer which specifies the maximum number of bytes which may be stored in the buffer.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
DATA_DECODE_BASE64	Decode a string that was created using the base64 encoding algorithm. This is the encoding method that is used by most modern email client software. Note that this option cannot be combined with other decoding methods.
DATA_DECODE_QUOTED	Decode a string that was created using the quoted-printable encoding algorithm. Note that this option cannot be combined with the other decoding methods.
DATA_DECODE_URL	Decode a string that was created using the URL encoding algorithm. Note that this option cannot be combined with the other decoding methods.
DATA_DECODE_UTF7	Decode a string that was created using the UTF-7 encoding algorithm. Note that this option cannot be combined with the other decoding methods.
DATA_DECODE_UTF8	Decode a string that was created using the UTF-8 encoding algorithm. Note that this option cannot be combined with the other decoding methods.
DATA_DECODE_COMPRESSED	The data was compressed prior to being encoded, and should be expanded after the decoding has completed successfully. This option is ignored if the

encoding type is not base64. This should only be used if it was specified when the data was encoded.

Return Value

If the function succeeds, it will return the number of bytes decoded and stored in the buffer. If the function fails, it will return -1. Failure typically indicates that the encoded string was corrupted or the buffer is not large enough to store the decoded data.

Remarks

The **DecodeBuffer** function is used to decode a block of data that was previously encoded with the **EncodeBuffer** function. To decode the contents of a file, it is recommended that you use the **DecodeFile** function instead.

If you specify either UTF-7 or UTF-8 encoding, the *lpOutput* parameter must point to a Unicode string and the *cbOutput* parameter must specify the maximum number of characters, not bytes, that can be copied into the string. The return value will be the number of Unicode characters, not bytes, that were copied into the output buffer. The conversion is always performed using the default system code page.

It is recommended that you use the **UnicodeEncodeText** function to convert a string to UTF-8 encoded text, and the **UnicodeDecodeText** function to convert UTF-8 encoded text to a UTF-16 or localized multi-byte string.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csncdv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DecodeFile](#), [EncodeBuffer](#), [EncodeFile](#), [GetMessageDigest](#), [IsUnicodeText](#), [UnicodeDecodeText](#), [UnicodeEncodeText](#)

DecodeFile Function

```
BOOL WINAPI DecodeFile(  
    LPCTSTR lpszInputFile,  
    LPCTSTR lpszOutputFile,  
    DWORD dwOptions,  
    DWORD dwReserved  
);
```

The **DecodeFile** function opens and decodes an encoded file, storing the contents in the specified file.

Parameters

lpszInputFile

A pointer to a string which specifies the name of the file to be decoded. The file must exist, and it must be a regular file that can be opened for reading by the current process. An error will be returned if a character device, such as CON: is specified as the file name.

lpszOutputFile

The name of the file that is to contain the decoded file data. If the file exists, it must be a regular file that can be opened for writing by the current process and will be overwritten. If the file does not exist, it will be created. An error will be returned if a character device, such as CON: is specified as the file name.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
FILE_DECODE_BASE64	Use the base64 algorithm for decoding the file. This is the encoding method that is used by most modern email client software. Note that this option cannot be combined with other decoding methods.
FILE_DECODE_QUOTED	Use the quoted-printable algorithm for decoding the file. Note that this option cannot be combined with the other decoding methods.
FILE_DECODE_UUCODE	Use the uuencode algorithm for decoding the file. This is a common encoding method used in UNIX systems and older email client software. Note that this option cannot be combined with other decoding methods.
FILE_DECODE_YENCODE	Use the yEnc algorithm for decoding the file. Note that this option cannot be combined with other decoding methods.
FILE_DECODE_COMPRESSED	The file was compressed prior to being encoded, and should be expanded after the decoding has completed successfully. This option should only be used if it was specified when the file was encoded.

dwReserved

This parameter is reserved and should always be set to zero.

Return Value

A non-zero value is returned if the file was successfully decoded. A zero value indicates that the file does not exist, the encoded file was damaged or the output file could not be created.

Remarks

The **DecodeFile** function decodes files that were previously encoded through a call to the **EncodeFile** function or by a third-party application such as an email client. The option to expand a previously compressed file requires that the function be able to create a temporary file on the local system in the directory specified by the TEMP environment variable. This function can only expand a file that was previously compressed with the **EncodeFile** or **CompressFile** functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csncdv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CompressFile](#), [EncodeFile](#), [ExpandFile](#), [GetMessageDigest](#)

EncodeBuffer Function

```
LONG WINAPI EncodeBuffer(  
    LPBYTE lpInput,  
    LONG cbInput,  
    LPTSTR lpszOutput,  
    LONG cchOutput,  
    DWORD dwOptions  
);
```

The **EncodeBuffer** function encodes the contents of the specified byte array, converting the data into printable text.

Parameters

lpInput

A pointer to a buffer that contains the data to be encoded.

cbInput

A long integer which specifies the number of characters in the buffer which should be encoded.

lpszOutput

A pointer to a string that will be used to store the encoded text. The length of the string must be at least 33% larger than the number of bytes being encoded, and will be terminated with a null byte.

cchOutput

The maximum number of characters that may be stored in the string buffer. If this value is too small to store the encoded text, the function will fail.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
DATA_ENCODE_BASE64	Use the base64 algorithm for encoding the data. This is the encoding method that is used by most modern email client software. Note that this option cannot be combined with other encoding methods.
DATA_ENCODE_QUOTED	Use the quoted-printable algorithm for encoding the data. Most printable characters are left as-is, with control characters and 8-bit characters encoded as their hexadecimal value. Note that this option cannot be combined with the other encoding methods.
DATA_ENCODE_URL	Use the URL encoding algorithm for encoding the data. Letters and numbers are left as-is, with control characters, most punctuation and 8-bit characters encoded as their hexadecimal value. Note that this option cannot be combined with the other encoding methods.
DATA_ENCODE_UTF7	Encode the data using the Unicode Transformation Format. This converts Unicode text into 7-bit

	characters. Note that this option cannot be combined with the other encoding methods.
DATA_ENCODE_UTF8	Encode the data using the Unicode Transformation Format. This converts Unicode text into 8-bit characters. Note that this option cannot be combined with the other encoding methods.
DATA_ENCODE_COMPRESSED	The data should be compressed before it is encoded. To restore the original data, it must be expanded after it has been decoded. This option is ignored if the encoding type is not base64.
DATA_ENCODE_LINEBREAK	The encoded data should be broken into multiple lines of text if the resulting string is longer than 72 characters. This option is ignored if the encoding type is not base64 or quoted-printable. This option should be specified if the encoded data is going to be included in an email message.

Return Value

If the function succeeds, it will return the number of bytes encoded and stored in the string. If the function fails, it will return -1. Failure typically indicates that the buffer is not large enough to store the encoded data.

Remarks

The **EncodeBuffer** function is used to encode a block of data and store it in the specified string buffer as printable text. To encode the contents of a file it is recommended that you use the **EncodeFile** function instead.

A common use of this function is to use the Base64 algorithm to obscure a plain text string. This technique is used by some Internet application protocols when passing authentication information over a standard connection. Although it is not a secure method of encrypting data, it does prevent a casual observer from reading the encoded text.

If you specify either UTF-7 or UTF-8 encoding, the *lpInput* parameter must point to a Unicode string and the *cbInput* parameter must specify the number of characters, not bytes, that are to be encoded. The conversion is always performed using the default system code page.

It is recommended that you use the **UnicodeEncodeText** function to convert a string to UTF-8 encoded text, and the **UnicodeDecodeText** function to convert UTF-8 encoded text to a UTF-16 or localized multi-byte string.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csncdv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DecodeBuffer](#), [DecodeFile](#), [EncodeFile](#), [GetMessageDigest](#), [IsUnicodeText](#), [UnicodeDecodeText](#), [UnicodeEncodeText](#)

EncodeFile Function

```
BOOL WINAPI EncodeFile(  
    LPCTSTR lpszInputFile,  
    LPCTSTR lpszOutputFile,  
    DWORD dwOptions,  
    DWORD dwReserved  
);
```

The **EncodeFile** function opens and encodes a file, storing the contents as printable text in the specified file.

Parameters

lpszInputFile

A pointer to a string which specifies the name of the file to be encoded. The file must exist, and it must be a regular file that can be opened for reading by the current process. An error will be returned if a character device, such as CON: is specified as the file name.

lpszOutputFile

The name of the file that is to contain the encoded file data. If the file exists, it must be a regular file that can be opened for writing by the current process and will be overwritten. If the file does not exist, it will be created. An error will be returned if a character device, such as CON: is specified as the file name.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
FILE_ENCODE_BASE64	Use the base64 algorithm for encoding the file. This is the encoding method that is used by most modern email client software. Note that this option cannot be combined with the other encoding methods.
FILE_ENCODE_UUCODE	Use the uuencode algorithm for encoding the file. This is a common encoding method used UNIX systems and older email client software. Note that this option cannot be combined with the other encoding methods.
FILE_ENCODE_QUOTED	Use the quoted-printable algorithm for encoding the file. Note that this option cannot be combined with the other encoding methods.
FILE_ENCODE_YENCODE	Use the yEnc algorithm for encoding the file. This is an encoding method that is commonly used when posting files to Usenet newsgroups. Note that this option cannot be combined with other encoding methods.
FILE_ENCODE_COMPRESSED	The file should be compressed before it is encoded. To restore the original contents of the file, it must be expanded after it has been decoded.

dwReserved

This parameter is reserved and should always be set to zero.

Return Value

A non-zero value is returned if the file was successfully encoded. A zero value indicates that the input file does not exist or the output file could not be created.

Remarks

The **EncodeFile** function converts binary data files to a format that contains only printable ASCII characters. The option to compress the file requires that the function be able to create a temporary file on the local system in the directory specified by the TEMP environment variables. A compressed file must be expanded with the **DecodeFile** or **ExpandFile** functions. The compressed file is not stored in an archive format that is recognized by third-party applications such as PKZip or WinZip.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csncdv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CompressFile](#), [DecodeFile](#), [ExpandFile](#), [GetMessageDigest](#)

ExpandBuffer Function

```
BOOL WINAPI ExpandBuffer(  
    LPVOID lpvInputBuffer,  
    DWORD cbInputBuffer  
    LPVOID lpvOutputBuffer,  
    LPDWORD lpcbOutputBuffer  
);
```

The **ExpandBuffer** function expands the contents of the previously compressed buffer.

Parameters

lpvInputBuffer

A pointer to the buffer which contains the compressed data.

cbInputBuffer

The number of bytes in the input buffer. This value must be greater than zero.

lpvOutputBuffer

A pointer to the buffer which will contain the expanded data. This parameter may be NULL, which can be used to determine the size of the of the resulting expanded data prior to allocating memory for it.

lpcbOutputBuffer

A pointer to the number of bytes that may be copied into the output buffer. If the *lpvOutputBuffer* parameter is not NULL, this must be initialized to a non-zero value. When the function returns, the actual number of bytes of expanded data is returned in this parameter.

Return Value

A non-zero value is returned if the data was successfully expanded. A zero value indicates that the data could not be expanded. Failure typically indicates that the compressed data was not in a recognized format.

Remarks

The compressed data buffer passed to the **ExpandBuffer** function must have been compressed by either the **CompressBuffer** or **CompressFile** functions. Data read from files that were compressed using third-party utilities such as WinZip will not be recognized by this function.

Example

```
cbOutputBuffer = 0;  
bResult = ExpandBuffer(lpInputBuffer,  
                        cbInputBuffer,  
                        NULL,  
                        &cbOutputBuffer);  
  
if (bResult)  
{  
    lpOutputBuffer = (LPBYTE)LocalAlloc(LPTR, cbOutputBuffer);  
    bResult = ExpandBuffer(lpInputBuffer,  
                           cbInputBuffer,  
                           lpOutputBuffer,  
                           &cbOutputBuffer);  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csncdv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CompressBuffer](#), [CompressFile](#), [DecodeFile](#), [EncodeFile](#), [ExpandFile](#)

ExpandFile Function

```
BOOL WINAPI ExpandFile(  
    LPCTSTR lpszInputFile,  
    LPCTSTR lpszOutputFile  
);
```

The **ExpandFile** function expands the contents of a previously compressed file.

Parameters

lpszInputFile

A pointer to a string which specifies the name of the file to be compressed. The file must exist, and it must be a regular file that can be opened for reading by the current process. An error will be returned if a character device, such as CON:, is specified as the file name.

lpszOutputFile

The name of the file that is to contain the expanded file data. If the file exists, it must be a regular file that can be opened for writing by the current process and will be overwritten. If the file does not exist, it will be created. An error will be returned if a character device, such as CON:, is specified as the file name.

Return Value

A non-zero value is returned if the file was successfully expanded. A zero value indicates that the input file could not be read or that the output file could not be created.

Remarks

The **ExpandFile** function can only expand files that were previously compressed using the **CompressFile** function. It cannot expand the contents of a file stored in an archive format used by third-party applications such as PKZip or WinZip.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csncdv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CompressFile](#), [DecodeFile](#), [EncodeFile](#)

ExpandFileEx Function

```
BOOL WINAPI ExpandFileEx(  
    LPCTSTR lpszInputFile,  
    LPCTSTR lpszOutputFile,  
    DWORD dwCompressionType,  
    LPVOID lpvHeaderBlock,  
    DWORD dwHeaderLength  
);
```

The **ExpandFileEx** function expands the contents of the previously compressed file.

Parameters

lpszInputFile

A pointer to a string which specifies the name of the file which contains the compressed data. The file must exist, and it must be a regular file that can be opened for reading by the current process. An error will be returned if a character device, such as CON: is specified as the file name.

lpszOutputFile

The name of the file that is to contain the expanded data. If the file exists, it must be a regular file that can be opened for writing by the current process and will be overwritten. If the file does not exist, it will be created. An error will be returned if a character device, such as CON: is specified as the file name.

dwCompressionType

A numeric value which determines the algorithm that was used to compress the data. One of the following values may be specified. By default, the Deflate algorithm is used.

Value	Constant	Description
1	COMPRESSION_TYPE_DEFLATE	A compression algorithm that combines LZ77 algorithm for creating common substrings and Huffman coding to process the different frequencies of byte sequences in the data stream. Deflate is widely used by compression software.
2	COMPRESSION_TYPE_BURROWSWHEELER	A compression algorithm that rearranges blocks of data in sorted order and then uses Huffman coding to process different frequencies of data within the block. Burrows-Wheeler compression provides a better compression ratio

		than the Deflate algorithm, however it requires more resources to perform the compression.
--	--	--

lpvHeaderBlock

A pointer to a buffer which will contain uncompressed data that is read from the beginning of the input file. If this parameter is NULL then no header block is read from the file.

dwHeaderLength

The length of the data buffer to be read from the beginning of the input file. If this value is zero, the *lpvHeaderBlock* parameter is ignored and no header block is read from the file.

Return Value

A non-zero value is returned if the file was successfully expanded. A zero value indicates that the input file could not be read or that the output file could not be created.

Remarks

The **ExpandFileEx** function can only expand files that were previously compressed using the **CompressFile** or **CompressFileEx** functions. It cannot expand the contents of a file stored in an archive format used by third-party applications such as PKZip or WinZip.

The *lpvHeaderBlock* and *dwHeaderLength* parameters can be used to be read application-defined data from the beginning of the file, commonly called a header block. Typically this is a fixed-length structure which provides information to the application about the compressed file.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csncdv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[CompressBuffer](#), [CompressFile](#), [DecodeFile](#), [EncodeFile](#), [ExpandBuffer](#), [ExpandFile](#)

GetMessageDigest Function

```
BOOL WINAPI GetMessageDigest(  
    LPVOID lpvBuffer,  
    DWORD cbBuffer,  
    DWORD dwDigestOptions,  
    LPTSTR lpszDigest,  
    INT nMaxLength  
);
```

The **GetMessageDigest** function computes a message digest for a buffer, string or data file.

Parameters

lpvBuffer

A pointer to the data that will be processed by the message digest algorithm. This parameter may be NULL, in which case the standard null hash value will be returned. Depending on the options specified, this may point to a block of memory, a null-terminated string or the name of a data file.

cbBuffer

The number of bytes of data that will be processed by the message digest algorithm. If this value is zero, then the standard null hash value will be returned. If the *lpvBuffer* parameter is NULL, this value must be zero. If this value is -1 then *lpvBuffer* is considered to be a pointer to a null-terminated string.

dwDigestOptions

A numeric value which specifies the algorithm that will be used to generate the message digest, and may be combined with other option flags. The algorithm may be one of the following values:

Value	Constant	Description
0	MESSAGE_DIGEST_DEFAULT	Use the default message digest algorithm, which is the same as specifying MESSAGE_DIGEST_SHA256
1	MESSAGE_DIGEST_MD5	Use the MD5 algorithm which generates a 128-bit hash value that is returned as as 32 digit hexadecimal string. This algorithm is typically used as a checksum to verify data integrity, however it is not considered cryptographically secure and hash collision is possible.
2	MESSAGE_DIGEST_SHA1	Use the SHA1 algorithm which generates a 160-bit hash value that is returned as a 40 digit hexadecimal string. This algorithm is typically used as a checksum to verify data integrity, however it is not considered cryptographically secure.
3	MESSAGE_DIGEST_SHA256	Use the SHA-256 algorithm which generates a 256-bit hash value that is returned as a 64 digit hexadecimal string. This algorithm is used for data integrity validation and various

		cryptographic uses, such as password storage and document signing.
4	MESSAGE_DIGEST_SHA384	Use the SHA-384 algorithm which generates a 384-bit hash value that is returned as a 96 digit hexadecimal string. This is a truncated version of the SHA-512 algorithm that uses different initial values.
5	MESSAGE_DIGEST_SHA512	Use the SHA-512 algorithm which generates a 512-bit hash value that is returned as a 128 digit hexadecimal string. This algorithm is used for data integrity validation and various cryptographic uses, such as password storage and document signing.

The following values may be combined with the algorithm ID to specify options that determine how the message digest is computed and returned to the caller:

Value	Constant	Description
0x10000	MESSAGE_DIGEST_MEMORY	The <i>lpvBuffer</i> parameter is a pointer to a block of memory and the <i>cbBuffer</i> parameter specifies how many bytes in the buffer should be used to compute the message digest. This is the default behavior if no other options are specified and the <i>cbBuffer</i> parameter is not -1.
0x20000	MESSAGE_DIGEST_STRING	The <i>lpvBuffer</i> parameter is a pointer to a null-terminated string and the <i>cbBuffer</i> parameter specifies how many characters in the string should be used to compute the message digest. If the <i>cbBuffer</i> parameter is -1 then all characters up to the terminating null byte will be used.
0x40000	MESSAGE_DIGEST_DATAFILE	The <i>lpvBuffer</i> parameter is a pointer to a null-terminated string that specifies the name of a data file, and the entire contents of the file will be used to compute the message digest. The <i>cbBuffer</i> parameter is ignored.
0x80000	MESSAGE_DIGEST_BASE64	The message digest will be returned in <i>lpzDigest</i> as a base64 encoded value rather than a string of hexadecimal digits. This can be useful for services that use HMAC message digests and require the hash to be in base64 format.

lpzDigest

A pointer to a string that will contain the message digest computed using the contents of the specified buffer. The digest value will consist of upper-case hexadecimal numbers, with the length varying based on the algorithm selected. The string buffer must be large enough to

accommodate the entire message digest, including the terminating null character. This parameter cannot be NULL.

nMaxLength

The maximum number of characters that may be copied into the *lpzDigest* string buffer, including the terminating null character. This value cannot be zero, and must be large enough to accommodate the complete message digest string value. It is recommended that the string buffer be at least MAX_DIGEST_STRING characters in size.

Return Value

A non-zero value is returned if the message digest was computed and returned in the string buffer provided by the caller. A zero value indicates that the digest could not be computed, or the string buffer provided by the caller was not large enough to contain the message digest value.

Remarks

The **GetMessageDigest** function accepts an arbitrary block of data, a null-terminated string or a data file and computes a fixed-length value (hash) that can be used to uniquely identify that content. These algorithms are designed to be one-way functions that are deterministic, have a low probability of collision and are difficult or infeasible to reverse. It should be noted that the MD5 and SHA1 algorithms are no longer considered cryptographically secure, however they are still widely used for data integrity validation to protect against accidental or unintended changes.

If the MESSAGE_DIGEST_STRING option is specified and the Unicode version of this function is called, the string will be automatically converted from UTF-16 to UTF-8 encoding, and the UTF-8 encoded text will be used to generate the message digest. If you need to generate a message digest using the UTF-16 encoded version of the string, use the MESSAGE_DIGEST_MEMORY option instead. Keep in mind that UTF-16 encoding represents each character with two bytes, so the buffer length should be twice the number of characters in the Unicode string.

If the MESSAGE_DIGEST_DATAFILE option is specified, the file contents will be processed as binary data with no special consideration given for the various end-of-line conventions used in text files, or any Unicode byte order marks (BOMs) in the file. It is permissible for the file to be empty (zero length), in which case the default null-hash value will be returned based on the algorithm selected by the caller.

The current thread will block while the message digest is being computed. If a large amount of data is being processed, this may cause the application to become non-responsive. In that case, it is recommended that you create a background worker thread and call this function from that worker thread, rather than from the main UI thread.

The length of the message digest string returned by this function will always be the same for a given algorithm, regardless of the size of the data buffer. For example, the length of the message digest for the SHA-256 algorithm will always be 64 characters long. Each byte of the message digest is represented by two upper-case hexadecimal numbers.

It is permitted to provide a NULL or zero-length data buffer to this function, in which case the standard null-hash value will be returned. The same null-hash value will also be returned for a zero-length string or an empty file.

Example

```
// Get a SHA-256 message digest for a string value
LPCTSTR lpzValue = _T("The quick brown fox jumps over the lazy dog");
TCHAR szDigest[MAX_DIGEST_STRING];
BOOL bSuccess;
```

```
bSuccess = GetMessageDigest(  
    lpszValue,  
    (DWORD)-1L,  
    MESSAGE_DIGEST_SHA256 | MESSAGE_DIGEST_STRING,  
    szDigest,  
    MAX_DIGEST_STRING);  
  
// Get an MD5 message digest for the contents of a data file  
LPCTSTR lpszFileName = _T("testfile.zip");  
  
bSuccess = GetMessageDigest(  
    lpszFileName,  
    0,  
    MESSAGE_DIGEST_MD5 | MESSAGE_DIGEST_DATAFILE,  
    szDigest,  
    MAX_DIGEST_STRING);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csncdv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[DecodeBuffer](#), [DecodeFile](#), [EncodeBuffer](#), [EncodeFile](#)

IsUnicodeText Function

```
BOOL WINAPI IsUnicodeText(  
    LPCTSTR lpString,  
    INT nLength  
);
```

The **IsUnicodeText** function checks if a string contains valid Unicode characters.

Parameters

lpString

A pointer to a null terminated string. This parameter cannot be NULL.

nLength

An integer value which specifies the number of characters to check. If this value is -1, the length of the string is determined by counting the number of characters up to the terminating null character. This parameter cannot be zero.

Return Value

A non-zero value is returned if the string contains valid Unicode characters. A zero value indicates the string contains characters that are not valid Unicode, or the string does not contain any Unicode characters.

Remarks

There are two versions of this function, **IsUnicodeTextA** which checks a multi-byte string to ensure that it contains valid UTF-8 encoded text, and **IsUnicodeTextW** which checks a wide string to ensure it contains valid UTF-16 text.

If the value of the *nLength* parameter is larger than the number of characters in the string, the function will not check beyond the terminating null character. If the length of the string is unknown, specify a length of -1 and the function will check the entire contents of the string up to the terminating null character.

If there is a byte order mark (BOM) sequence at the beginning of the string, this will be recognized by the function. A string that contains a valid BOM sequence with no corresponding text will be successfully validated by this function.

This function does not perform checks to ensure the string contains printable characters. It only validates that the Unicode string is structurally valid. The **IsUnicodeTextA** function will check to make sure there are no invalid UTF-8 encodings. The **IsUnicodeTextW** function will check to make sure the UTF-16 string does not contain any unpaired surrogates.

This function does not preserve state information and cannot be used to check the validity of a stream of text or binary data. It should only be used to validate complete Unicode strings that are terminated with a null character.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csncdv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

UnicodeDecodeText Function

```
INT WINAPI UnicodeDecodeText(  
    UINT nCodePage,  
    LPCSTR lpUtf8Text,  
    INT nLength,  
    LPTSTR lpString,  
    INT nMaxLength  
);
```

The **UnicodeDecodeText** function decodes UTF-8 encoded text and returns the contents in a string buffer.

Parameters

nCodePage

A value that specifies the code page used when decoding the text. This parameter can be set to the value of any code page that is available in the operating system. It is recommended that you use CP_ACP which specifies the system default ANSI code page. This parameter is only used when decoding UTF-8 encoded text to a multi-byte string.

lpUtf8Text

A pointer to a null terminated string that contains the UTF-8 encoded text to be decoded. This parameter must always specify a pointer to an 8-bit character string, regardless if the project is configured to use the Unicode or Multi-Byte character set. This parameter cannot be NULL.

nLength

The number of bytes in the UTF-8 encoded string to be decoded. If this parameter is -1, the length of the UTF-8 encoded text is determined by counting the number of characters up to the terminating null character.

lpString

A pointer to a string buffer that will contain the decoded UTF-8 text when the function returns. The string buffer must be large enough to contain all of the encoded text, and cannot be a NULL a pointer.

nMaxLength

The maximum number of characters that can be copied into the string buffer. The contents of the *lpString* buffer will always be null terminated, and the maximum size must be large enough to include the null character. This value must be greater than zero.

Return Value

If the UTF-8 encoded text is successfully decoded, the return value is the number of characters copied to the string buffer. If the text cannot be decoded, or the string buffer is not large enough to store all of the decoded text, the function will return zero. To get extended error information, call **GetLastError**.

Remarks

There are two versions of this function, **UnicodeDecodeTextA** which returns a localized multi-byte string and **UnicodeDecodeTextW** which converts the UTF-8 encoded text to UTF-16 text. Your project configuration typically determines which version of this function is used by default.

If the value of the *nLength* parameter is larger than the number of UTF-8 characters, the function will not check beyond the terminating null character. If the length of the *lpUtf8Text* string is

unknown, specify a length of -1 and the function will decode the entire contents of the string up to the terminating null character.

When calling **UnicodeDecodeTextA** to convert UTF-8 encoded text to a localized multi-byte string, it is recommended that you specify CP_ACP (zero) as the code page value unless you know it contains Unicode characters that cannot be represented using the default ANSI code page. Using CP_ACP will ensure the UTF-8 text is decoded using the current locale and language settings.

When calling **UnicodeDecodeTextW** to convert UTF-8 encoded text to UTF-16 text, the code page parameter is ignored and should always be a value of zero.

This function performs a strict check on the UTF-8 encoded text and will fail if the encoding is malformed, or in the case of being converted to a multi-byte string, if it cannot be decoded using the specified code page. It will not simply replace invalid character sequences with a default character.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csncdv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IsUnicodeText](#), [UnicodeEncodeText](#)

UnicodeEncodeText Function

```
INT WINAPI UnicodeEncodeText(  
    UINT nCodePage,  
    LPCTSTR lpString,  
    INT nLength,  
    LPSTR lpUtf8Text,  
    INT nMaxLength  
);
```

The **UnicodeEncodeText** function encodes a string and returns UTF-8 encoded text.

Parameters

nCodePage

A value that specifies the code page used when encoding the text. This parameter can be set to the value of any code page that is available in the operating system. It is recommended that you use CP_ACP which specifies the system default ANSI code page. This parameter is only used when encoding a multi-byte string.

lpString

A pointer to a null terminated string that contains the text to be encoded. This parameter cannot be NULL.

nLength

The number of characters in the string to be encoded. If this parameter is -1, the length of the string is determined by counting the number of characters up to the terminating null character.

lpUtf8Text

A pointer to a character buffer which will contain the UTF-8 encoded text when the function returns. This parameter must always specify a pointer to an 8-bit character string, regardless if the project is configured to use the Unicode or Multi-Byte character set. This parameter cannot be NULL.

nMaxLength

The maximum number of bytes that can be copied into the UTF-8 text buffer. The contents of the *lpUtf8Text* buffer will always be null terminated, and the maximum size must be large enough to include the null byte. This value must be greater than zero.

Return Value

If the string is successfully encoded, the return value is the number of characters copied to the output buffer. If the text cannot be encoded, or the output buffer is not large enough to store all of the encoded text, the function will return zero. To get extended error information, call **GetLastError**.

Remarks

There are two versions of this function, **UnicodeEncodeTextA** which converts a multi-byte string to UTF-8 encoded text, and **UnicodeEncodeTextW** which converts a UTF-16 string to a UTF-8 string. Your project configuration typically determines which version of this function is used by default.

If the value of the *nLength* parameter is larger than the number of characters in *lpString*, the function will not check beyond the terminating null character. If the length of *lpString* is unknown, specify a length of -1 and the function will encode the entire contents of the string up to the

terminating null character.

When calling **UnicodeEncodeTextA** to convert a multi-byte string to UTF-8 encoding, it is recommended that you specify CP_ACP (zero) as the code page value unless you know it contains ANSI characters from a different code page. Using CP_ACP will ensure the string is encoded using the current locale and language settings.

This function performs a strict check on the multi-byte input string and will fail if it contains a malformed multi-byte sequence or characters that cannot be converted to UTF-8 using the specified code page. It will not simply replace invalid character sequences with a default character.

When calling **UnicodeEncodeTextW** to convert UTF-16 text to UTF-8 encoded text, the code page parameter is ignored and should always be a value of zero. The text will be normalized prior to being converted to UTF-8 using canonical composition, where decomposed characters are combined to create their canonical precomposed equivalent.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csncdv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IsUnicodeText](#), [UnicodeDecodeText](#)

File Transfer Protocol Client Library

Transfer files between a local and server and perform common file management functions on the server.

Reference

- [Functions](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

File Name	CSFTPV10.DLL
Version	10.0.1468.2518
LibID	C01B14F0-7091-429D-A690-B99D635CB8AB
Import Library	CSFTPV10.LIB
Dependencies	None
Standards	RFC 959, RFC 1579, RFC 2228

Overview

The File Transfer Protocol (FTP) library provides a comprehensive API which supports both high level operations, such as uploading or downloading files, as well as a collection of lower-level file I/O functions. In addition to file transfers, an application can create, rename and delete files and directories, search for files using wildcards and perform other common file management functions.

Files can be stored on the local file system or in memory, depending on the needs of your application and multiple file transfers be performed using a single function call. The library can also be used to manage files on the server and supports many of the common protocol extensions that can be used to access the remote file system. It understands a number of different directory listing formats, including those typically used with UNIX and Linux based systems, Windows server platforms, NetWare servers and VMS systems.

This library supports active and passive mode file transfers, firewall compatibility options, proxy servers and secure file transfers using the standard SSL/TLS and SFTP protocols. Secure file transfers support implicit and explicit SSL sessions, client certificates and up to 256-bit AES encryption.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

File Transfer Protocol Functions

Function	Description
FtpAllocate	Allocate the specified number of bytes on the server
FtpAsyncConnect	Connect asynchronously to the specified server
FtpAsyncFileList	Return an unparsed list of files in the specified directory
FtpAsyncGetData	Copy the contents of a remote file to a local buffer
FtpAsyncGetFile	Copy a file from the server to the local system
FtpAsyncGetFileEx	Copy a file from the server to the local system, supporting files larger than 4GB
FtpAsyncProxyConnect	Connect asynchronously to the specified proxy server
FtpAsyncPutData	Create a file on the server using the contents of a local buffer
FtpAsyncPutFile	Copy a file from the local system to the server
FtpAsyncPutFileEx	Copy a file from the local system to the server, supporting files larger than 4GB
FtpAttachThread	Attach the specified client handle to another thread
FtpCancel	Cancel the current blocking operation
FtpChangeDirectory	Change the current working directory on the server
FtpChangeDirectoryUp	Change the current working directory on the server
FtpCloseDirectory	Close the open directory on the server
FtpCloseFile	Close the current file on the server
FtpCommand	Send a command to the server
FtpConnect	Establish a client connection with a server
FtpConnectUrl	Establish a client connection using the specified URL
FtpCreateDirectory	Create the specified directory on the server
FtpCreateSecurityCredentials	Create a new security credentials structure
FtpDeleteFile	Delete a file from the server
FtpDeleteSecurityCredentials	Delete a previously created security credentials structure
FtpDisableEvents	Disable event notification
FtpDisableTrace	Disable logging of socket function calls to the trace log
FtpDisconnect	Disconnect the client session from the server
FtpDownloadFile	Download a file from the server to the local system
FtpEnableEvents	Enable event notification
FtpEnableFeature	Enable the specified feature in the client
FtpEnableTrace	Enable logging of socket function calls to a file

FtpEndOfFile	Determine if the end-of-file has been reached
FtpEnumFiles	Return a list of files in a directory on the server
FtpEnumFilesEx	Return a list of files in a directory on the server, supporting files larger than 4GB
FtpEnumTasks	Return a list of asynchronous tasks
FtpEventProc	Callback function that processes events generated by the client
FtpFreezeEvents	Suspend and resume event handling by the client
FtpGetActivePorts	Return the range of local port numbers used for active transfers
FtpGetBufferSize	Return the size of an internal buffer used during data transfers
FtpGetChannelMode	Return the mode for the specified communication channel
FtpGetClientQuota	Return quota information for the current client session
FtpGetData	Copy the contents of a remote file to a local buffer
FtpGetDirectory	Get the current working directory on the server
FtpGetDirectoryFormat	Get the format which is used by the server to list files
FtpGetErrorString	Return a description for the specified error code
FtpGetFeatures	Return the features available to the client
FtpGetFile	Copy a file from the server to the local system
FtpGetFileEx	Copy a file from the server to the local system, supporting files larger than 4GB
FtpGetFileList	Return an unparsed list of files in the specified directory
FtpGetFileNameEncoding	Return the character encoding used when sending a file name to the server
FtpGetFilePermissions	Return the access permissions for the specified file
FtpGetFileSize	Return the size of a file on the server
FtpGetFileStatus	Return file status information from the server
FtpGetFileStatusEx	Return file status information from the server, supporting files larger than 4GB
FtpGetFileTime	Return the modification time for the specified file on the server
FtpGetFileType	Return the default file type for the current session
FtpGetFirstFile	Return the first file from the file list returned by the server
FtpGetFirstFileEx	Return the first file from the file list returned by the server, supporting files larger than 4GB
FtpGetLastError	Return the last error code
FtpGetMultipleFiles	Copy multiple files from the server to the local system
FtpGetNextFile	Return the next file from the file list returned by the server
FtpGetNextFileEx	Return the next file from the file list returned by the server, supporting files larger than 4GB

FtpGetPriority	Return the current priority for file transfers
FtpGetProxyType	Return the proxy type selected by the client
FtpGetResultCode	Return the result code from the previous command
FtpGetResultString	Return the result string from the previous command
FtpGetSecurityInformation	Return security information about the current client connection
FtpGetServerInformation	Get system information about the server
FtpGetServerStatus	Return system status of server
FtpGetServerTimeZone	Return the timezone offset in seconds for the current server
FtpGetServerType	Return the type of operating system the server is running on
FtpGetStatus	Return the current client status
FtpGetTaskError	Return the last error code for the specified asynchronous task
FtpGetTaskId	Return the unique task identifier associated with the specified client session
FtpGetText	Download the contents of a text file to a string buffer
FtpGetTimeout	Return the number of seconds until an operation times out
FtpGetTransferStatus	Return current file transfer statistics
FtpGetTransferStatusEx	Return current file transfer statistics, supporting files larger than 4GB
FtpInitialize	Initialize the library and validate the specified runtime license key
FtpIsBlocking	Determine if the current operation is blocked
FtpIsConnected	Determine if the client is connected to the server
FtpIsReadable	Determine if the client can read data from the data channel
FtpIsWritable	Determine if the client can write data to the data channel
FtpLogin	Login to the server
FtpLogout	Logout from the server
FtpMountStructure	Mount a structure (filesystem) on the server
FtpOpenDirectory	Open the specified directory for reading
FtpOpenFile	Open the specified file for reading or writing
FtpProxyConnect	Establish a connection with a proxy server
FtpPutData	Create a file on the server using the contents of a local buffer
FtpPutFile	Copy a file from the local system to the server
FtpPutFileEx	Copy a file from the local system to the server, supporting files larger than 4GB
FtpPutMultipleFiles	Copy multiple files from the local system to the server
FtpPutText	Create a text file on the server from the contents of a string buffer
FtpRead	Read data from the server

FtpRegisterEvent	Register an event handler for the specified event
FtpRegisterFileType	Associate a file name extension with a specific file type
FtpRemoveDirectory	Remove a directory from the server
FtpRenameFile	Rename a file on the server
FtpReset	Reset the client connection
FtpSetActivePorts	Set the range of local port numbers used for active transfers
FtpSetBufferSize	Set the size of an internal buffer used during data transfers
FtpSetChannelMode	Change the security mode for the specified channel
FtpSetDirectoryFormat	Set the format which is used by the server to list files
FtpSetFeatures	Set the features which can be used by the client
FtpSetFileMode	Set the current file mode
FtpSetFileNameEncoding	Set the character encoding type used when sending a file name to the server
FtpSetFilePermissions	Set the access permissions for the specified file
FtpSetFileStructure	Set the current file data structure
FtpSetFileTime	Set the modification time for the specified file on the server
FtpSetFileType	Set the default file type for the current session
FtpSetLastError	Set the last error code
FtpSetPassiveMode	Set the server in passive mode
FtpSetPriority	Set the priority for file transfers
FtpSetTimeout	Set the number of seconds until an operation times out
FtpTaskAbort	Abort the specified asynchronous task
FtpTaskDone	Determine if an asynchronous task has completed
FtpTaskResume	Resume execution of an asynchronous task
FtpTaskSuspend	Suspend execution of an asynchronous task
FtpTaskWait	Wait for an asynchronous task to complete
FtpUninitialize	Terminate use of the library by the calling process
FtpUploadFile	Upload a file from the local system to the server
FtpValidateUrl	Check the contents of a string to ensure it represents a valid URL
FtpVerifyFile	Compare the contents of a local file against a file stored on the server
FtpWrite	Write data to the server

FtpAllocate Function

```
INT WINAPI FtpAllocate(  
    HCLIENT hClient,  
    DWORD dwFileLength,  
    DWORD dwRecSize  
);
```

The **FtpAllocate** function instructs the server to reserve sufficient storage to accommodate the new file being transferred.

Parameters

hClient

Handle to the client session.

dwFileLength

The number of bytes to allocate storage for on the server.

dwRecSize

The maximum record or page size for the file. A value of zero indicates that the file does not have a record or page structure, and the parameter is ignored.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function should be called immediately before the **FtpOpenFile** function.

This function is ignored by those servers which do not require that the maximum size of the file be declared beforehand. The most common FTP servers running under UNIX and Windows do not require that file space be pre-allocated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

See Also

[FtpOpenFile](#), [FtpSetFileMode](#), [FtpSetFileStructure](#), [FtpSetFileType](#)

FtpAsyncConnect Function

```
HCLIENT WINAPI FtpAsyncConnect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSECURITYCREDENTIALS LpCredentials,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **FtpAsyncConnect** function establishes a connection with the specified server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **FtpConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 21. For secure connections, the default port number is 990. If the secure port number is specified, an implicit SSL/TLS connection will be established by default.

lpszUserName

Points to a string that specifies the user name to be used to authenticate the current client session. If this parameter is NULL or an empty string, then the login is considered to be anonymous. Note that anonymous logins are not supported for secure connections using the SSH protocol.

lpszPassword

Points to a string that specifies the password to be used to authenticate the current client session. This parameter may be NULL or an empty string if no password is required for the specified user, or if no username has been specified.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:



Constant	Description
FTP_OPTION_PASSIVE	<p>This option specifies the client should attempt to establish the data connection with the server. When the client uploads or downloads a file, normally the server establishes a second connection back to the client which is used to transfer the file data.</p> <p>However, if the local system is behind a firewall or a NAT router, the server may not be able to create the data connection and the transfer will fail. By specifying this option, it forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.</p>
FTP_OPTION_FIREWALL	<p>This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.</p>
FTP_OPTION_NOAUTH	<p>This option specifies the server does not require authentication, or that it requires an alternate authentication method. When this option is used, the client connection is flagged as authenticated as soon as the connection to the server has been established. Note that using this option to bypass authentication may result in subsequent errors when attempting to retrieve a directory listing or transfer a file. It is recommended that you consult the technical reference documentation for the server to determine its specific authentication requirements.</p>
FTP_OPTION_KEEPALIVE	<p>This option specifies the client should attempt to keep the connection with the server active for an extended period of time. It is important to note that regardless of this option, the server may still choose to disconnect client sessions that are holding the command channel open but are not performing file transfers.</p>
FTP_OPTION_NOAUTHRSA	<p>This option specifies that RSA authentication should not be used with SSH-1 connections. This option is ignored with SSH-2 connections and should only be specified if required by the server. This option has no effect on standard or secure connections using SSL.</p>

FTP_OPTION_NOPWDNUL	This option specifies the user password cannot be terminated with a null character. This option is ignored with SSH-2 connections and should only be specified if required by the server. This option has no effect on standard or secure connections using SSL.
FTP_OPTION_NOREKEY	This option specifies the client should never attempt a repeat key exchange with the server. Some SSH servers do not support rekeying the session, and this can cause the client to become non-responsive or abort the connection after being connected for an hour. This option has no effect on standard or secure connections using SSL.
FTP_OPTION_COMPATSID	This compatibility option changes how the session ID is handled during public key authentication with older SSH servers. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions. This option has no effect on standard or secure connections using SSL.
FTP_OPTION_COMPATHMAC	This compatibility option changes how the HMAC authentication codes are generated. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions. This option has no effect on standard or secure connections using SSL.
FTP_OPTION_VIRTUALHOST	This option specifies the server supports virtual hosting, where multiple domains are hosted by a server using the same external IP address. If this option is enabled, the client will send the HOST command to the server upon establishing a connection.
FTP_OPTION_VERIFY	This option specifies that file transfers should be automatically verified after the transfer has completed. If the server supports the XMD5 command, the transfer will be verified by calculating an MD5 hash of the file contents. If the server does not support the XMD5 command, but does support the XCRC command, the transfer will be verified by calculating a CRC32 checksum of the file contents. If neither the XMD5 or XCRC commands are supported, the transfer is verified by comparing the size of the file. Automatic file verification is only performed for binary mode transfers because of the end-of-line conversion that may occur when text files are uploaded or downloaded.
FTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server

	certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. This option is the same as specifying FTP_OPTION_SECURE_IMPLICIT which immediately performs the SSL/TLS protocol negotiation when the connection is established.
FTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to immediately establish secure SSL/TLS connection with the server. This option is typically used when connecting to a server on port 990, which is the default port number used for FTPS.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should establish a standard connection to the server and then use the AUTH command to negotiate an explicit secure connection. This option is typically used when connecting to the server on ports other than 990.
FTP_OPTION_SECURE_SHELL	This option specifies the client should use the Secure Shell (SSH) protocol to establish the connection. This option will automatically be selected if the connection is established using port 22, the default port for SSH.
FTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
FTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established. This option also forces all connections to be outbound and enables the firewall compatibility features in the client.
FTP_OPTION_KEEPALIVE_DATA	This option specifies the client should attempt to keep the control connection active during a file transfer. Normally, when a data transfer is in progress, no additional commands are issued on the control channel until the transfer completes. Specifying this option automatically enables the FTP_OPTION_KEEPALIVE option and forces the client to continue to issue NOOP commands during the file transfer. This option only applies to FTP and

	FTPS connections and has no effect on connections using SFTP (SSH).
FTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
FTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
FTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller files being uploaded or downloaded using fast network connections.
FTP_OPTION_TLS_REUSE	This option specifies that TLS session reuse should be enabled for secure connections. This option is only supported on Windows 8.1 or Windows Server 2012 R2 and later platforms, and it should only be used when explicitly required by the server. This option is not compatible with servers built using OpenSSL 1.0.2 and earlier versions which do not provide Extended Master Secret (EMS) support as outlined in RFC7627.

lpCredentials

A pointer to a [SECURITYCREDENTIALS](#) structure. This parameter should be NULL if the connection is not secure or when client credentials are not required. Most servers do not require a client certificate to establish a secure connection. However, if the server does require a client certificate, the structure members *dwSize*, *lpzCertStore* and *lpzCertName* must be defined. Undefined structure members must be initialized to a value of zero or NULL and the *dwSize* member must be initialized to the size of the **SECURITYCREDENTIALS** structure.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **FtpGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
FTP_EVENT_CONNECT	The control connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
FTP_EVENT_DISCONNECT	The server has closed the control connection to the client. The client should read any remaining data and disconnect.
FTP_EVENT_OPENFILE	The data connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
FTP_EVENT_CLOSEFILE	The server has closed the data connection to the client. The client should read any remaining data and close the data channel.
FTP_EVENT_READFILE	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_WRITEFILE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
FTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and reconnect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
FTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
FTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.
FTP_EVENT_GETFILE	This event is generated when a file download has completed. If multiple files are being downloaded, this event will be generated for each file.

FTP_EVENT_PUTFILE	This event is generated when a file upload has completed. If multiple files are being uploaded, this event will be generated for each file.
-------------------	---

To cancel asynchronous notification and return the client to a blocking mode, use the **FtpDisableEvents** function.

If the FTP_OPTION_KEEPALIVE option is specified, a background worker thread will be created to monitor the command channel and periodically send NOOP commands to the server if no commands have been sent recently. This can prevent the server from terminating the client connection during idle periods where no commands are being issued. However, it is important to keep in mind that many servers can be configured to also limit the total amount of time a client can be connected to the server, as well as the amount of time permitted between file transfers. If the server does not respond to the NOOP command, this option will be automatically disabled for the remainder of the client session.

If the FTP_OPTION_SECURE_EXPLICIT option is specified, the client will establish a standard connection to the server and send the AUTH TLS command to the server. If the server does not accept this command, it will then send the AUTH SSL command. If both commands are rejected by the server, an explicit SSL session cannot be established. By default, both the command and data channels will be encrypted when a secure connection is established. To change this, use the **FtpSetChannelMode** function.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **FtpAttachThread** function.

Specifying the FTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Example

```
// The Ipswitch WS_FTP server accepts the AUTH command to establish
// an explicit SSL session on the default FTP port
hClient = FtpAsyncConnect(lpszRemoteHost,
                        FTP_PORT_DEFAULT,
                        FTP_TIMEOUT,
                        FTP_OPTION_SECURE_EXPLICIT,
                        NULL,
                        hEventWnd,
                        uEventMsg);

// When the GlobalSCAPE Secure FTP server is configured in implicit
// authorization mode, it negotiates a secure session as soon as the
// connection is established and does not require a command
hClient = FtpAsyncConnect(lpszRemoteHost,
                        FTP_PORT_SECURE,
                        FTP_TIMEOUT,
                        FTP_OPTION_SECURE_IMPLICIT,
```

```
NULL,  
hEventWnd,  
uEventMsg);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpAsyncProxyConnect](#), [FtpConnect](#), [FtpCreateSecurityCredentials](#), [FtpDeleteSecurityCredentials](#), [FtpDisconnect](#), [FtpGetSecurityInformation](#), [FtpInitialize](#), [FtpProxyConnect](#), [FtpSetChannelMode](#)

FtpAsyncFileList Function

```
UINT WINAPI FtpAsyncFileList(  
    HCLIENT hClient,  
    LPCTSTR lpszDirectory,  
    DWORD dwOptions,  
    LPTSTR lpszBuffer,  
    INT nMaxLength,  
    FTPEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

Return an unparsed list of files in the specified directory.

Parameters

hClient

Handle to the client session.

lpszDirectory

A pointer to a string that specifies the name of a directory and/or a wildcard file mask. The format of the directory name must match the file naming conventions of the server. If this parameter is NULL or points to an empty string, the current working directory will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Constant	Description
FTP_LIST_DEFAULT	This option specifies the server should return a complete listing of files in the specified directory with as much detail as possible. This typically means that the file size, date, ownership and access rights will be returned to the client. Information about the files are returned in lines of text, with each line terminated by carriage return and linefeed (CRLF) characters. The exact format of the data returned is specific to the server operating system.
FTP_LIST_NAMEONLY	This option specifies the server should only return a list of file names, with no additional information about the file. Each file name is terminated by carriage return and linefeed (CRLF) characters.

lpszBuffer

A pointer to a string buffer that will contain the list of files when the function returns. This buffer should be large enough to store the complete file listing and a terminating null character. If the buffer is smaller than the total amount of data returned by the server, the data will be truncated. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **FtpGetLastError** function.

Remarks

The **FtpAsyncFileList** function returns a list of files in the specified directory, copying the data to a string buffer. This function is similar to the **FtpGetFileList** function, however it retrieves the list of files asynchronously using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file listing is being returned by the server.

Because this function works asynchronously, it is important that the memory allocated for the buffer is not released before the asynchronous task completes. If you provide a buffer that is allocated on the stack, such as with the example listed below, then you must ensure that your code does not return from the function while the directory listing is being retrieved. In the example, this is achieved by calling the **FtpTaskWait** function. You can also perform other operation and poll the status of the task by calling the **FtpTaskDone** function. If you wish to return from the calling function immediately, then you must dynamically allocate memory for the *lpzBuffer* parameter on the heap and free that memory after the task has completed and the data is no longer needed.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the FTP_EVENT_CONNECT event after the connection has been established, and the FTP_EVENT_DISCONNECT event after the directory listing has completed. This enables your application to know when the directory listing is about to begin, and immediately before the worker thread is terminated. The worker thread creates a secondary connection to the server with its own session handle. This ensures that the asynchronous operation will not interfere with the current client session. Your application can interact with this background worker thread using the client handle that is passed to the event handler.

This function can be particularly useful when the client is connected to a server that returns file listings in a format that is not recognized by the library. The application can retrieve the unparsed file listing from the server and parse the contents. Note that if you specify the FTP_LIST_NAMEONLY option, the data will only contain a list of file names and there will be no way for the application to know if they represent a regular file or a subdirectory.

This function is supported for both FTP and SFTP (SSH) connections, however the format of the data may differ depending on which protocol is used. Most UNIX based FTP servers will not list files and subdirectories that begin with a period, however most SFTP servers will return a list of all files, even those that begin with a period.

Example

```
TCHAR szFileList[MAXFILELISTSIZE];  
UINT nTaskId;
```

```
nTaskId = FtpAsyncFileList(hClient,
                           NULL,
                           FTP_LIST_DEFAULT,
                           szFileList,
                           MAXFILELISTSIZE,
                           NULL, 0);

if (nTaskId == 0)
{
    _tprintf(_T("Unable to list files (error 0x%08lx)\n"), FtpGetLastError());
    return;
}
else
{
    DWORD dwError = NO_ERROR;
    DWORD dwElapsed = 0;

    FtpTaskWait(nTaskId, INFINITE, &dwElapsed, &dwError);
    _tprintf(_T("Asynchronous file listing returned in %lu milliseconds\n"),
dwElapsed);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpEventProc](#), [FtpGetFileList](#), [FtpTaskWait](#)

FtpAsyncGetData Function

```
UINT WINAPI FtpAsyncGetData(  
    HCLIENT hClient,  
    LPCTSTR lpszRemoteFile,  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwReserved,  
    FTPEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

Copies the contents of a file on the server to the specified buffer.

Parameters

hClient

Handle to the client session.

lpszRemoteFile

A pointer to a string that specifies the file on the server that will be transferred to the local system. The file naming conventions must be that of the host operating system.

lpvBuffer

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvBuffer* parameter. If the *lpvBuffer* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the file that was downloaded.

dwReserved

A reserved parameter. This value should always be zero.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **FtpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **FtpAsyncGetData** function is used to download the contents of a remote file into a local

buffer. This function is similar to the **FtpGetData** function, however it retrieves the contents of the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being copied to the local system.

Because this function works asynchronously, it is important that the memory allocated for the buffer is not released before the asynchronous task completes. If you provide a buffer that is allocated on the stack, such as with the example listed below, then you must ensure that your code does not return from the function while the data is being downloaded. In the example, this is achieved by calling the **FtpTaskWait** function. You can also perform other operation and poll the status of the task by calling the **FtpTaskDone** function. If you wish to return from the calling function immediately, then you must dynamically allocate memory for the *lpvBuffer* and *lpdwLength* parameters on the heap and free that memory after the task has completed and the data is no longer needed.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the FTP_EVENT_CONNECT event after the connection has been established, and the FTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the data transfer is about to begin, and immediately before the worker thread is terminated. The worker thread creates a secondary connection to the server with its own session handle. This ensures that the asynchronous operation will not interfere with the current client session. Your application can interact with this background worker thread using the client handle that is passed to the event handler.

The *lpvBuffer* parameter may be specified in one of two ways, depending on the needs of the application. It can either be a pre-allocated buffer large enough to store the contents of the file or it can specify the address of a global memory handle that will contain the data. If it points to a pre-allocated buffer, the *lpdwLength* parameter must be initialized to the maximum number of bytes that can be copied into the buffer. If specifies the address of a global memory handle, then *lpdwLength* must be initialized to a value of zero. See the example code below.

Example

```
HGLOBAL hglobalBuffer = NULL;
DWORD dwLength = 0;
UINT nTaskId;

// Return the file data into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hglobalBuffer parameter

nTaskId = FtpAsyncGetData(hClient,
                        lpzFileName,
                        &hglobalBuffer,
                        &dwLength,
                        0, NULL, 0);

if (nTaskId != 0)
{
    DWORD dwError = NO_ERROR;
    DWORD dwElapsed = 0;

    // Wait for the transfer to complete
    FtpTaskWait(nTaskId, INFINITE, &dwElapsed, &dwError);

    // Lock the global memory handle, returning a pointer to the
```

```
// contents of the file data
lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

// After the data has been used, the handle must be unlocked
// and freed, otherwise a memory leak will occur
GlobalUnlock(hgblBuffer);
GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpAsyncGetFile](#), [FtpAsyncPutData](#), [FtpAsyncPutFile](#), [FtpEventProc](#), [FtpGetFile](#), [FtpTaskWait](#)

FtpAsyncGetFile Function

```
UINT WINAPI FtpAsyncGetFile(  
    HCLIENT hClient,  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszRemoteFile,  
    DWORD dwOptions,  
    DWORD dwOffset,  
    FTPEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

Downloads the specified file from the server to the local system.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A pointer to a string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A pointer to a string that specifies the file on the server that will be transferred to the local system. The file naming conventions must be that of the host operating system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Constant	Description
FTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the local file exists, it will be overwritten with the contents of the downloaded file.
FTP_TRANSFER_APPEND	This option specifies that if the local file exists, the contents of file on the server is appended to the local file. If the local file does not exist, it is created.

dwOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **FtpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **FtpAsyncGetFile** function is used to download the contents of a remote file to a file on the local system. This function is similar to the **FtpGetFile** function, however it retrieves the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being transferred to the local system.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the `FTP_EVENT_CONNECT` event after the connection has been established, and the `FTP_EVENT_DISCONNECT` event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated. During the file transfer, the callback function will be invoked periodically with the `FTP_EVENT_PROGRESS` event. The client session handle is passed to the event handler, allowing you to call functions such as **FtpGetTransferStatus** to determine the amount of data that has been copied.

To determine when the transfer has completed without implementing an event handler, periodically call the **FtpTaskDone** function. If you wish to block the current thread and wait for the transfer to complete, call the **FtpTaskWait** function. To stop a background file transfer that is in progress, call the **FtpTaskAbort** function. This will signal the background worker thread to cancel the transfer and terminate the session.

This function can be called multiple times to download multiple files in the background; however, most servers limit the number of simultaneous connections that can originate from a single IP address. It is recommended that you only perform two simultaneous background transfers from the same server at any one time. The application should not make any assumptions about the order in which multiple background transfers may complete or how they are sequenced. For example, it should never be assumed that a background task with a lower task ID will complete before a task with a higher ID value.

Example

```
UINT nTaskId;

// Begin a file transfer in the background

nTaskId = FtpAsyncGetFile(hClient,
                        lpszLocalFile,
                        lpszRemoteFile,
                        FTP_TRANSFER_DEFAULT,
                        0,
                        NULL,
                        0);

if (nTaskId != 0)
{
    DWORD dwError = NO_ERROR;
    DWORD dwElapsed = 0;
```

```
// Wait for the transfer to complete
FtpTaskWait(nTaskId, INFINITE, &dwElapsed, &dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpAsyncGetData](#), [FtpAsyncPutData](#), [FtpAsyncPutFile](#), [FtpEventProc](#), [FtpGetFile](#), [FtpTaskDone](#),
[FtpTaskWait](#)

FtpAsyncGetFileEx Function

```
UINT WINAPI FtpAsyncGetFileEx(  
    HCLIENT hClient,  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszRemoteFile,  
    DWORD dwOptions,  
    ULARGE_INTEGER uiOffset,  
    FTPEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

Downloads the specified file from the server to the local system. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A pointer to a string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A pointer to a string that specifies the file on the server that will be transferred to the local system. The file naming conventions must be that of the host operating system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Constant	Description
FTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the local file exists, it will be overwritten with the contents of the downloaded file.
FTP_TRANSFER_APPEND	This option specifies that if the local file exists, the contents of file on the server is appended to the local file. If the local file does not exist, it is created.

uiOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **FtpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **FtpAsyncGetFileEx** function is used to download the contents of a remote file to a file on the local system. This function is similar to the **FtpGetFileEx** function, however it retrieves the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being transferred to the local system.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the `FTP_EVENT_CONNECT` event after the connection has been established, and the `FTP_EVENT_DISCONNECT` event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated. During the file transfer, the callback function will be invoked periodically with the `FTP_EVENT_PROGRESS` event. The client session handle is passed to the event handler, allowing you to call functions such as **FtpGetTransferStatusEx** to determine the amount of data that has been copied.

To determine when the transfer has completed without implementing an event handler, periodically call the **FtpTaskDone** function. If you wish to block the current thread and wait for the transfer to complete, call the **FtpTaskWait** function. To stop a background file transfer that is in progress, call the **FtpTaskAbort** function. This will signal the background worker thread to cancel the transfer and terminate the session.

This function can be called multiple times to download multiple files in the background; however, most servers limit the number of simultaneous connections that can originate from a single IP address. It is recommended that you only perform two simultaneous background transfers from the same server at any one time. The application should not make any assumptions about the order in which multiple background transfers may complete or how they are sequenced. For example, it should never be assumed that a background task with a lower task ID will complete before a task with a higher ID value.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpAsyncPutFileEx](#), [FtpEventProc](#), [FtpGetFileEx](#), [FtpPutFileEx](#), [FtpTaskDone](#), [FtpTaskWait](#)

FtpAsyncProxyConnect Function

```
HCLIENT WINAPI FtpAsyncProxyConnect(  
    UINT nProxyType,  
    LPCTSTR LpszProxyHost,  
    UINT nProxyPort,  
    LPCTSTR LpszProxyUser,  
    LPCTSTR LpszProxyPassword,  
    LPCTSTR LpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSECURITYCREDENTIALS lpCredentials,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **FtpAsyncProxyConnect** function establishes a connection with the specified server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **FtpProxyConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

nProxyType

An identifier which specifies the type of proxy server that is being connected to. This value must be defined as one of the following values:

Constant	Description
FTP_PROXY_NONE	This value specifies that no proxy server is being used. In this case, the FtpConnect function is called directly, ignoring the proxy parameters.
FTP_PROXY_USER	This value specifies that the client is not logged into the proxy server. The USER command is sent in the format username@ftpsite followed by the password. This is the format used with the Gauntlet proxy server.
FTP_PROXY_LOGIN	This value specifies that the client is logged into the proxy server. The USER command is then sent in the format username@ftpsite followed by the password. This is the format used by the InterLock proxy server.
FTP_PROXY_OPEN	This value specifies that the client is not logged into the proxy server. The OPEN command is sent specifying the host name, followed by the username and password.
FTP_PROXY_SITE	This value specifies that the client is logged into the server. The SITE command is sent, specifying the host name, followed by the

	username and the password.
FTP_PROXY_OTHER	This special proxy type specifies that another, undefined proxy server is being used. The client connects to the proxy host, but does not attempt to authenticate the client. The application is responsible for negotiating with the proxy server, typically using the FtpCommand function to send specific command sequences.

lpszProxyHost

A pointer to the name of the proxy server to connect through; this may be a fully-qualified domain name or an IP address.

lpszProxyPort

The port number the proxy server is listening on; a value of zero specifies that the default port number should be used.

lpszProxyUser

A pointer to the user name used to authenticate the client on the proxy server. Not all proxy servers require this information; it is recommended that you consult the proxy server documentation to determine if a username is required.

lpszProxyPassword

A pointer to the password used to authenticate the client on the proxy server. Not all proxy servers require this information; it is recommended that you consult the proxy server documentation to determine if a password is required.

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used. For standard connections, the default port number is 21. For secure connections, the default port number is 990. If the secure port number is specified, an implicit SSL/TLS connection will be established by default.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
FTP_OPTION_PASSIVE	This option specifies the client should attempt to establish the data connection with the server. When the client uploads or downloads a file, normally the server establishes a second connection back to the client which is used to transfer the file data. However, if the local system is behind a firewall or a NAT router, the server may not be able to create the data connection and the transfer will fail. By specifying this option, it forces the client to establish

	<p>an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.</p>
FTP_OPTION_FIREWALL	<p>This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.</p>
FTP_OPTION_NOAUTH	<p>This option specifies the server does not require authentication, or that it requires an alternate authentication method. When this option is used, the client connection is flagged as authenticated as soon as the connection to the server has been established. Note that using this option to bypass authentication may result in subsequent errors when attempting to retrieve a directory listing or transfer a file. It is recommended that you consult the technical reference documentation for the server to determine its specific authentication requirements.</p>
FTP_OPTION_KEEPAIVE	<p>This option specifies the client should attempt to keep the connection with the server active for an extended period of time. It is important to note that regardless of this option, the server may still choose to disconnect client sessions that are holding the command channel open but are not performing file transfers.</p>
FTP_OPTION_VIRTUALHOST	<p>This option specifies the server supports virtual hosting, where multiple domains are hosted by a server using the same external IP address. If this option is enabled, the client will send the HOST command to the server upon establishing a connection.</p>
FTP_OPTION_VERIFY	<p>This option specifies that file transfers should be automatically verified after the transfer has completed. If the server supports the XMD5 command, the transfer will be verified by calculating an MD5 hash of the file contents. If the server does not support the XMD5 command, but does support the XCRC command, the transfer will be verified by calculating a CRC32 checksum of the file contents. If neither the XMD5 or XCRC commands are supported, the transfer is verified by comparing the size of the file. Automatic file verification is only</p>

	performed for binary mode transfers because of the end-of-line conversion that may occur when text files are uploaded or downloaded.
FTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. This option is the same as specifying FTP_OPTION_SECURE_IMPLICIT which immediately performs the SSL/TLS protocol negotiation when the connection is established.
FTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to immediately establish secure SSL/TLS connection with the server. This option is typically used when connecting to a server on port 990, which is the default port number used for FTPS.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should establish a standard connection to the server and then use the AUTH command to negotiate an explicit secure connection. This option is typically used when connecting to the server on ports other than 990.
FTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
FTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established. This option also forces all connections to be outbound and enables the firewall compatibility features in the client.
FTP_OPTION_KEEPALIVE_DATA	This option specifies the client should attempt to keep the control connection active during a file transfer. Normally, when a data transfer is in progress, no additional commands are issued on the control channel until the transfer completes. Specifying this option automatically enables the FTP_OPTION_KEEPALIVE option and forces the client to continue to issue NOOP commands during the file transfer. This option only applies to FTP and FTPS connections and has no effect on connections

	using SFTP (SSH).
FTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
FTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
FTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller files being uploaded or downloaded using fast network connections.
FTP_OPTION_TLS_REUSE	This option specifies that TLS session reuse should be enabled for secure connections. This option is only supported on Windows 8.1 or Windows Server 2012 R2 and later platforms, and it should only be used when explicitly required by the server. This option is not compatible with servers built using OpenSSL 1.0.2 and earlier versions which do not provide Extended Master Secret (EMS) support as outlined in RFC7627.

lpCredentials

A pointer to a [SECURITYCREDENTIALS](#) structure. This parameter should be NULL if the connection is not secure or when client credentials are not required. Most servers do not require a client certificate to establish a secure connection. However, if the server does require a client certificate, the structure members *dwSize*, *lpzCertStore* and *lpzCertName* must be defined. Undefined structure members must be initialized to a value of zero or NULL and the *dwSize* member must be initialized to the size of the **SECURITYCREDENTIALS** structure.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **FtpGetLastError**.

Remarks

The username and password that is used to authenticate the client with the proxy server are not the same as those used to login to the target server. Once a connection has been established with the proxy server, the client must call the **FtpLogin** function to actually login to the server and begin a file transfer.

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
FTP_EVENT_CONNECT	The control connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
FTP_EVENT_DISCONNECT	The server has closed the control connection to the client. The client should read any remaining data and disconnect.
FTP_EVENT_OPENFILE	The data connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
FTP_EVENT_CLOSEFILE	The server has closed the data connection to the client. The client should read any remaining data and close the data channel.
FTP_EVENT_READFILE	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_WRITEFILE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
FTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
FTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
FTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a

	status control or progress bar.
FTP_EVENT_GETFILE	This event is generated when a file download has completed. If multiple files are being downloaded, this event will be generated for each file.
FTP_EVENT_PUTFILE	This event is generated when a file upload has completed. If multiple files are being uploaded, this event will be generated for each file.

To cancel asynchronous notification and return the client to a blocking mode, use the **FtpDisableEvents** function.

If the FTP_OPTION_KEEPALIVE option is specified, a background worker thread will be created to monitor the command channel and periodically send NOOP commands to the server if no commands have been sent recently. This can prevent the server from terminating the client connection during idle periods where no commands are being issued. However, it is important to keep in mind that many servers can be configured to also limit the total amount of time a client can be connected to the server, as well as the amount of time permitted between file transfers. If the server does not respond to the NOOP command, this option will be automatically disabled for the remainder of the client session.

If the FTP_OPTION_SECURE_EXPLICIT option is specified, the client will establish a standard connection to the server and send the AUTH TLS command to the server. If the server does not accept this command, it will then send the AUTH SSL command. If both commands are rejected by the server, an explicit SSL session cannot be established. By default, both the command and data channels will be encrypted when a secure connection is established. To change this, use the **FtpSetChannelMode** function.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **FtpAttachThread** function.

Specifying the FTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpAsyncConnect](#), [FtpConnect](#), [FtpCreateSecurityCredentials](#), [FtpDeleteSecurityCredentials](#), [FtpDisconnect](#), [FtpGetSecurityInformation](#), [FtpInitialize](#), [FtpLogin](#), [FtpProxyConnect](#), [FtpSetChannelMode](#)

FtpAsyncPutData Function

```
UINT WINAPI FtpAsyncPutData(  
    HCLIENT hClient,  
    LPCTSTR lpszFileName,  
    LPVOID lpvBuffer,  
    DWORD dwLength,  
    DWORD dwReserved,  
    FTPEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

Copies the contents of the specified buffer to a file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

A pointer to a string that specifies the file on the server that will be created, overwritten or appended to. The file naming conventions must be that of the host operating system.

lpvBuffer

A pointer to the data that will be copied to the server and stored in the specified file.

dwLength

The number of bytes to copy from the buffer.

dwReserved

A reserved parameter. This value should always be zero.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **FtpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **FtpAsyncPutData** function is used to upload the contents of a local buffer to the server. This function is similar to the **FtpPutData** function, however it uses a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the data is being sent to the server.

Because this function works asynchronously, it is important that the memory allocated for the buffer is not released before the asynchronous task completes. If you provide a buffer that is

allocated on the stack, ensure that your code does not return from the function while the data is being uploaded. This can be achieved by calling the **FtpTaskWait** function or periodically calling the **FtpTaskDone** function to determine if the transfer has completed. If you wish to return from the calling function immediately, then you must dynamically allocate memory for the *lpvBuffer* parameter on the heap and free that memory after the task has completed and the data is no longer needed.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the FTP_EVENT_CONNECT event after the connection has been established, and the FTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the data transfer is about to begin, and immediately before the worker thread is terminated. The worker thread creates a secondary connection to the server with its own session handle. This ensures that the asynchronous operation will not interfere with the current client session. Your application can interact with this background worker thread using the client handle that is passed to the event handler.

If the *lpvBuffer* parameter is pointing to a Unicode string, it is important to note that the value of the *dwLength* parameter should specify the number of bytes, not the number of characters. When using UTF-16, each character is two bytes long and therefore the length of the buffer is effectively double the length of the string. Because Unicode strings can contain null characters, you must also set the current file type to FILE_TYPE_IMAGE prior to calling this function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpAsyncGetData](#), [FtpAsyncGetFile](#), [FtpAsyncPutFile](#), [FtpEventProc](#), [FtpPutData](#), [FtpTaskWait](#)

FtpAsyncPutFile Function

```
UINT WINAPI FtpAsyncPutFile(  
    HCLIENT hClient,  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszRemoteFile,  
    DWORD dwOptions,  
    DWORD dwOffset,  
    FTPEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

Uploads the specified file from the local system to the server.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A pointer to a string that specifies the file that will be transferred from the local system. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A pointer to a string that specifies the file on the server that will be created, overwritten or appended to. The file naming conventions must be that of the host operating system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Constant	Description
FTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the remote file exists, it will be overwritten with the contents of the uploaded file.
FTP_TRANSFER_APPEND	This option specifies that if the remote file exists, the contents of the local file is appended to the remote file. If the remote file does not exist, it is created.

dwOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **FtpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **FtpAsyncPutFile** function is used to upload the contents of a local file to the server. This function is similar to the **FtpPutFile** function, however it uploads the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being transferred to the server.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the `FTP_EVENT_CONNECT` event after the connection has been established, and the `FTP_EVENT_DISCONNECT` event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated. During the file transfer, the callback function will be invoked periodically with the `FTP_EVENT_PROGRESS` event. The client session handle is passed to the event handler, allowing you to call functions such as **FtpGetTransferStatus** to determine the amount of data that has been copied.

To determine when the transfer has completed without implementing an event handler, periodically call the **FtpTaskDone** function. If you wish to block the current thread and wait for the transfer to complete, call the **FtpTaskWait** function. To stop a background file transfer that is in progress, call the **FtpTaskAbort** function. This will signal the background worker thread to cancel the transfer and terminate the session.

This function can be called multiple times to upload multiple files in the background; however, most servers limit the number of simultaneous connections that can originate from a single IP address. It is recommended that you only perform two simultaneous background transfers from the same server at any one time. The application should not make any assumptions about the order in which multiple background transfers may complete or how they are sequenced. For example, it should never be assumed that a background task with a lower task ID will complete before a task with a higher ID value.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpAsyncGetData](#), [FtpAsyncGetFile](#), [FtpAsyncPutData](#), [FtpEventProc](#), [FtpPutFile](#), [FtpTaskDone](#), [FtpTaskWait](#)

FtpAsyncPutFileEx Function

```
UINT WINAPI FtpAsyncPutFileEx(  
    HCLIENT hClient,  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszRemoteFile,  
    DWORD dwOptions,  
    ULARGE_INTEGER uiOffset,  
    FTPEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

Uploads the specified file from the local system to the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A pointer to a string that specifies the file that will be transferred from the local system. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A pointer to a string that specifies the file on the server that will be created, overwritten or appended to. The file naming conventions must be that of the host operating system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Constant	Description
FTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the remote file exists, it will be overwritten with the contents of the uploaded file.
FTP_TRANSFER_APPEND	This option specifies that if the remote file exists, the contents of the local file is appended to the remote file. If the remote file does not exist, it is created.

uiOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **FtpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **FtpAsyncPutFileEx** function is used to upload the contents of a local file to the server. This function is similar to the **FtpPutFileEx** function, however it uploads the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being transferred to the server.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the `FTP_EVENT_CONNECT` event after the connection has been established, and the `FTP_EVENT_DISCONNECT` event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated. During the file transfer, the callback function will be invoked periodically with the `FTP_EVENT_PROGRESS` event. The client session handle is passed to the event handler, allowing you to call functions such as **FtpGetTransferStatusEx** to determine the amount of data that has been copied.

To determine when the transfer has completed without implementing an event handler, periodically call the **FtpTaskDone** function. If you wish to block the current thread and wait for the transfer to complete, call the **FtpTaskWait** function. To stop a background file transfer that is in progress, call the **FtpTaskAbort** function. This will signal the background worker thread to cancel the transfer and terminate the session.

This function can be called multiple times to upload multiple files in the background; however, most servers limit the number of simultaneous connections that can originate from a single IP address. It is recommended that you only perform two simultaneous background transfers from the same server at any one time. The application should not make any assumptions about the order in which multiple background transfers may complete or how they are sequenced. For example, it should never be assumed that a background task with a lower task ID will complete before a task with a higher ID value.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpAsyncGetFileEx](#), [FtpEventProc](#), [FtpGetFileEx](#), [FtpPutFileEx](#), [FtpTaskDone](#), [FtpTaskWait](#)

FtpAttachThread Function

```
DWORD WINAPI FtpAttachThread(  
    HCLIENT hClient  
    DWORD dwThreadId  
);
```

The **FtpAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **FtpAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **FtpAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **FtpCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **FtpUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

See Also

[FtpCancel](#), [FtpAsyncConnect](#), [FtpConnect](#), [FtpDisconnect](#), [FtpUninitialize](#)

FtpCancel Function

```
INT WINAPI FtpCancel(  
    HCLIENT hClient  
);
```

The **FtpCancel** function cancels any outstanding blocking operation in the client, causing the blocking function to fail. The application may then retry the operation or terminate the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

When the **FtpCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

This function may be called during a blocking file transfer.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpIsBlocking](#)

FtpChangeDirectory Function

```
INT WINAPI FtpChangeDirectory(  
    HCLIENT hClient,  
    LPCTSTR lpszDirectory  
);
```

The **FtpChangeDirectory** function changes the current working directory for the client session.

Parameters

hClient

Handle to the client session.

lpszDirectory

Points to a string that specifies the name of the directory. The file pathing and name conventions must be that of the server.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function uses the CWD command to change the current working directory. The user must have the appropriate permission to access the specified directory.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpChangeDirectoryUp](#), [FtpCloseDirectory](#), [FtpCommand](#), [FtpGetDirectory](#), [FtpGetFirstFile](#), [FtpGetNextFile](#), [FtpGetResultCode](#), [FtpGetResultString](#), [FtpOpenDirectory](#)

FtpChangeDirectoryUp Function

```
INT WINAPI FtpChangeDirectoryUp(  
    HCLIENT hClient  
);
```

The **FtpChangeDirectoryUp** function changes directory to the parent of the current working directory.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function sends the CDUP command to the server. This command is a special case of the CWD command, and is included to simplify transferring between directory trees on those operating systems which have different syntaxes for naming the parent directory. The current user must have the appropriate permission to access the specified directory.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpChangeDirectory](#), [FtpCommand](#), [FtpGetDirectory](#), [FtpGetResultCode](#), [FtpGetResultString](#)

FtpCloseDirectory Function

```
INT WINAPI FtpCloseDirectory(  
    HCLIENT hClient  
);
```

The **FtpCloseDirectory** function closes the data socket connection to the server.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function must be called after all of the file information from the server has been returned. Because directory information is returned on the data channel, no file transfers can take place while a directory is being read.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[FtpGetDirectoryFormat](#), [FtpGetFileStatus](#), [FtpGetFirstFile](#), [FtpGetNextFile](#), [FtpOpenDirectory](#), [FtpSetDirectoryFormat](#)

FtpCloseFile Function

```
INT WINAPI FtpCloseFile(  
    HCLIENT hClient  
);
```

The **FtpCloseFile** function flushes the internal client buffers and closes the data socket connection to the server.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

If the file is opened for writing, all buffered data is written to the server before the socket is closed. This may cause the client to block until all of the data can be written. The client application should not perform any other action until the function returns.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

See Also

[FtpOpenFile](#), [FtpRead](#), [FtpWrite](#)

FtpCommand Function

```
INT WINAPI FtpCommand(  
    HCLIENT hClient,  
    LPCTSTR LpszCommand,  
    LPCTSTR LpszParameter  
);
```

The **FtpCommand** function sends a command to the server and returns the result code back to the caller. This function is typically used for site-specific commands not directly supported by the API.

Parameters

hClient

Handle to the client session.

lpszCommand

The command which will be executed by the server.

lpszParameter

An optional command parameter. If the command requires more than one parameter, then they should be combined into a single string, with a space separating each parameter. If the command does not accept any parameters, this value may be NULL.

Return Value

If the function succeeds, the return value is the result code returned by the server. If the function fails, the return value is `FTP_ERROR`. To get extended error information, call **FtpGetLastError**.

Remarks

This function should only be used when the application needs to send a custom, site-specific command or send a command that is not directly supported by this API. This function should never be used to issue a command that opens a data channel. If the application needs to transform data as it is being sent or received, and cannot use the **FtpGetFile** or **FtpPutFile** functions, then use the **FtpOpenFile** function to open a data channel with the server.

By default, file names which are sent to the server using the **FtpCommand** function are sent as ANSI characters. If the Unicode version of the function is used, the file name will be converted from Unicode to ANSI using the current codepage. If the server supports UTF-8 encoded file names, the **FtpSetFileNameEncoding** function can be used to specify that file names with non-ASCII characters should be sent as UTF-8 encoded values. It is important to note that this option is only available if the server advertises support for UTF-8 and permits that encoding type.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetFile](#), [FtpGetFileNameEncoding](#), [FtpGetResultCode](#), [FtpGetResultString](#), [FtpOpenFile](#), [FtpPutFile](#), [FtpSetFileNameEncoding](#)

FtpConnect Function

```
HCLIENT WINAPI FtpConnect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSECURITYCREDENTIALS LpCredentials  
);
```

The **FtpConnect** function establishes a connection with the server and defines security related options to be used.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used. For standard connections, the default port number is 21. For secure connections, the default port number is 990.

lpszUserName

Points to a string that specifies the user name to be used to authenticate the current client session. If this parameter is NULL or an empty string, then the login is considered to be anonymous. Note that anonymous logins are not supported for secure connections using the SSH protocol.

lpszPassword

Points to a string that specifies the password to be used to authenticate the current client session. This parameter may be NULL or an empty string if no password is required for the specified user, or if no username has been specified.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
FTP_OPTION_PASSIVE	This option specifies the client should attempt to establish the data connection with the server. When the client uploads or downloads a file, normally the server establishes a second connection back to the client which is used to transfer the file data. However, if the local system is behind a firewall or a NAT router, the server may not be able to create the data connection and the transfer will fail. By

	<p>specifying this option, it forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.</p>
FTP_OPTION_FIREWALL	<p>This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.</p>
FTP_OPTION_NOAUTH	<p>This option specifies the server does not require authentication, or that it requires an alternate authentication method. When this option is used, the client connection is flagged as authenticated as soon as the connection to the server has been established. Note that using this option to bypass authentication may result in subsequent errors when attempting to retrieve a directory listing or transfer a file. It is recommended that you consult the technical reference documentation for the server to determine its specific authentication requirements.</p>
FTP_OPTION_KEEPAIVE	<p>This option specifies the client should attempt to keep the connection with the server active for an extended period of time. It is important to note that regardless of this option, the server may still choose to disconnect client sessions that are holding the command channel open but are not performing file transfers.</p>
FTP_OPTION_NOAUTHRSA	<p>This option specifies that RSA authentication should not be used with SSH-1 connections. This option is ignored with SSH-2 connections and should only be specified if required by the server. This option has no effect on standard or secure connections using SSL.</p>
FTP_OPTION_NOPWDNUL	<p>This option specifies the user password cannot be terminated with a null character. This option is ignored with SSH-2 connections and should only be specified if required by the server. This option has no effect on standard or secure connections using SSL.</p>
FTP_OPTION_NOREKEY	<p>This option specifies the client should never attempt a repeat key exchange with the server. Some SSH servers do not support rekeying the session, and</p>

	<p>this can cause the client to become non-responsive or abort the connection after being connected for an hour. This option has no effect on standard or secure connections using SSL.</p>
FTP_OPTION_COMPATSID	<p>This compatibility option changes how the session ID is handled during public key authentication with older SSH servers. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions. This option has no effect on standard or secure connections using SSL.</p>
FTP_OPTION_COMPATHMAC	<p>This compatibility option changes how the HMAC authentication codes are generated. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions. This option has no effect on standard or secure connections using SSL.</p>
FTP_OPTION_VIRTUALHOST	<p>This option specifies the server supports virtual hosting, where multiple domains are hosted by a server using the same external IP address. If this option is enabled, the client will send the HOST command to the server upon establishing a connection.</p>
FTP_OPTION_VERIFY	<p>This option specifies that file transfers should be automatically verified after the transfer has completed. If the server supports the XMD5 command, the transfer will be verified by calculating an MD5 hash of the file contents. If the server does not support the XMD5 command, but does support the XCRC command, the transfer will be verified by calculating a CRC32 checksum of the file contents. If neither the XMD5 or XCRC commands are supported, the transfer is verified by comparing the size of the file. Automatic file verification is only performed for binary mode transfers because of the end-of-line conversion that may occur when text files are uploaded or downloaded.</p>
FTP_OPTION_TRUSTEDSITE	<p>This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.</p>
FTP_OPTION_SECURE	<p>This option specifies the client should attempt to establish a secure connection with the server. This option is the same as specifying FTP_OPTION_SECURE_IMPLICIT which immediately performs the SSL/TLS protocol negotiation when the connection is established.</p>

FTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to immediately establish secure SSL/TLS connection with the server. This option is typically used when connecting to a server on port 990, which is the default port number used for FTPS.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should establish a standard connection to the server and then use the AUTH command to negotiate an explicit secure connection. This option is typically used when connecting to the server on ports other than 990.
FTP_OPTION_SECURE_SHELL	This option specifies the client should use the Secure Shell (SSH) protocol to establish the connection. This option will automatically be selected if the connection is established using port 22, the default port for SSH.
FTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
FTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established. This option also forces all connections to be outbound and enables the firewall compatibility features in the client.
FTP_OPTION_KEEPALIVE_DATA	This option specifies the client should attempt to keep the control connection active during a file transfer. Normally, when a data transfer is in progress, no additional commands are issued on the control channel until the transfer completes. Specifying this option automatically enables the FTP_OPTION_KEEPALIVE option and forces the client to continue to issue NOOP commands during the file transfer. This option only applies to FTP and FTPS connections and has no effect on connections using SFTP (SSH).
FTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client

	will attempt to establish a connection using IPv6 regardless if this option has been specified.
FTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
FTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller files being uploaded or downloaded using fast network connections.
FTP_OPTION_TLS_REUSE	This option specifies that TLS session reuse should be enabled for secure connections. This option is only supported on Windows 8.1 or Windows Server 2012 R2 and later platforms, and it should only be used when explicitly required by the server. This option is not compatible with servers built using OpenSSL 1.0.2 and earlier versions which do not provide Extended Master Secret (EMS) support as outlined in RFC7627.

lpCredentials

A pointer to a [SECURITYCREDENTIALS](#) structure. This parameter should be NULL if the connection is not secure or when client credentials are not required. Most servers do not require a client certificate to establish a secure connection. However, if the server does require a client certificate, the structure members *dwSize*, *lpzCertStore* and *lpzCertName* must be defined. Undefined structure members must be initialized to a value of zero or NULL and the *dwSize* member must be initialized to the size of the **SECURITYCREDENTIALS** structure.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **FtpGetLastError**.

Remarks

To prevent this function from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **FtpConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

If the FTP_OPTION_KEEPALIVE option is specified, a background worker thread will be created to monitor the command channel and periodically send NOOP commands to the server if no commands have been sent recently. This can prevent the server from terminating the client connection during idle periods where no commands are being issued. However, it is important to keep in mind that many servers can be configured to also limit the total amount of time a client can be connected to the server, as well as the amount of time permitted between file transfers. If the server does not respond to the NOOP command, this option will be automatically disabled for the remainder of the client session.

If the FTP_OPTION_SECURE_EXPLICIT option is specified, the client will establish a standard

connection to the server and send the AUTH TLS command to the server. If the server does not accept this command, it will then send the AUTH SSL command. If both commands are rejected by the server, an explicit SSL session cannot be established. By default, both the command and data channels will be encrypted when a secure connection is established. To change this, use the **FtpSetChannelMode** function.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **FtpAttachThread** function.

Specifying the FTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Example

```
// Connect to a server using a standard (non-secure) connection on
// the default port. The username and password are not encrypted.

hClient = FtpConnect(lpszRemoteHost,
                    FTP_PORT_DEFAULT,
                    lpszUserName,
                    lpszPassword,
                    FTP_TIMEOUT,
                    FTP_OPTION_DEFAULT,
                    NULL);

// Connect to a server using the default port and then initiate a
// secure connection using the AUTH TLS command

hClient = FtpConnect(lpszRemoteHost,
                    FTP_PORT_DEFAULT,
                    lpszUserName,
                    lpszPassword,
                    FTP_TIMEOUT,
                    FTP_OPTION_PASSIVE | FTP_OPTION_SECURE_EXPLICIT,
                    NULL);

// Connect to a server on port 990 and immediately initiate a
// secure connection as soon as the connection is established

hClient = FtpConnect(lpszRemoteHost,
                    FTP_PORT_SECURE,
                    lpszUserName,
                    lpszPassword,
                    FTP_TIMEOUT,
                    FTP_OPTION_PASSIVE | FTP_OPTION_SECURE_IMPLICIT,
                    NULL);

// Connect to a server on port 22 using the Secure Shell (SFTP)
// protocol to transfer files

hClient = FtpConnect(lpszRemoteHost,
```

```
FTP_PORT_SSH,  
lpszUserName,  
lpszPassword,  
FTP_TIMEOUT,  
FTP_OPTION_SECURE_SHELL,  
NULL);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpCreateSecurityCredentials](#), [FtpDeleteSecurityCredentials](#), [FtpDisconnect](#),
[FtpGetSecurityInformation](#), [FtpInitialize](#), [FtpProxyConnect](#), [FtpSetChannelMode](#)

FtpConnectUrl Function

```
HCLIENT WINAPI FtpConnectUrl(  
    LPCTSTR lpszURL,  
    UINT nTimeout,  
    DWORD dwOptions  
);
```

The **FtpConnectUrl** function establishes a connection with the specified server using a URL.

Parameters

lpszURL

A pointer to a string which specifies the URL for the server. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
FTP_OPTION_PASSIVE	This option specifies the client should attempt to establish the data connection with the server. When the client uploads or downloads a file, normally the server establishes a second connection back to the client which is used to transfer the file data. However, if the local system is behind a firewall or a NAT router, the server may not be able to create the data connection and the transfer will fail. By specifying this option, it forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_NOAUTH	This option specifies the server does not require authentication, or that it requires an alternate authentication method. When this option is used, the client connection is flagged as authenticated as

	<p>soon as the connection to the server has been established. Note that using this option to bypass authentication may result in subsequent errors when attempting to retrieve a directory listing or transfer a file. It is recommended that you consult the technical reference documentation for the server to determine its specific authentication requirements.</p>
FTP_OPTION_KEEPALIVE	<p>This option specifies the client should attempt to keep the connection with the server active for an extended period of time. It is important to note that regardless of this option, the server may still choose to disconnect client sessions that are holding the command channel open but are not performing file transfers.</p>
FTP_OPTION_NOAUTHRSA	<p>This option specifies that RSA authentication should not be used with SSH-1 connections. This option is ignored with SSH-2 connections and should only be specified if required by the server. This option has no effect on standard or secure connections using SSL.</p>
FTP_OPTION_NOPWDNUL	<p>This option specifies the user password cannot be terminated with a null character. This option is ignored with SSH-2 connections and should only be specified if required by the server. This option has no effect on standard or secure connections using SSL.</p>
FTP_OPTION_NOREKEY	<p>This option specifies the client should never attempt a repeat key exchange with the server. Some SSH servers do not support rekeying the session, and this can cause the client to become non-responsive or abort the connection after being connected for an hour. This option has no effect on standard or secure connections using SSL.</p>
FTP_OPTION_COMPATSID	<p>This compatibility option changes how the session ID is handled during public key authentication with older SSH servers. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions. This option has no effect on standard or secure connections using SSL.</p>
FTP_OPTION_COMPATHMAC	<p>This compatibility option changes how the HMAC authentication codes are generated. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions. This option has no effect on standard or secure connections using SSL.</p>

FTP_OPTION_VIRTUALHOST	This option specifies the server supports virtual hosting, where multiple domains are hosted by a server using the same external IP address. If this option is enabled, the client will send the HOST command to the server upon establishing a connection.
FTP_OPTION_VERIFY	This option specifies that file transfers should be automatically verified after the transfer has completed. If the server supports the XMD5 command, the transfer will be verified by calculating an MD5 hash of the file contents. If the server does not support the XMD5 command, but does support the XCRC command, the transfer will be verified by calculating a CRC32 checksum of the file contents. If neither the XMD5 or XCRC commands are supported, the transfer is verified by comparing the size of the file. Automatic file verification is only performed for binary mode transfers because of the end-of-line conversion that may occur when text files are uploaded or downloaded.
FTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. This option is the same as specifying FTP_OPTION_SECURE_IMPLICIT which immediately performs the SSL/TLS protocol negotiation when the connection is established.
FTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to immediately establish secure SSL/TLS connection with the server. This option is typically used when connecting to a server on port 990, which is the default port number used for FTPS.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should establish a standard connection to the server and then use the AUTH command to negotiate an explicit secure connection. This option is typically used when connecting to the server on ports other than 990.
FTP_OPTION_SECURE_SHELL	This option specifies the client should use the Secure Shell (SSH) protocol to establish the connection. This option will automatically be selected if the connection is established using port 22, the default port for SSH.
FTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility

	with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
FTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established. This option also forces all connections to be outbound and enables the firewall compatibility features in the client.
FTP_OPTION_KEEPALIVE_DATA	This option specifies the client should attempt to keep the control connection active during a file transfer. Normally, when a data transfer is in progress, no additional commands are issued on the control channel until the transfer completes. Specifying this option automatically enables the FTP_OPTION_KEEPALIVE option and forces the client to continue to issue NOOP commands during the file transfer. This option only applies to FTP and FTPS connections and has no effect on connections using SFTP (SSH).
FTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
FTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
FTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller files being uploaded or downloaded using fast network connections.
FTP_OPTION_TLS_REUSE	This option specifies that TLS session reuse should be enabled for secure connections. This option is only supported on Windows 8.1 or Windows Server 2012 R2 and later platforms, and it should only be used when explicitly required by the server. This

option is not compatible with servers built using OpenSSL 1.0.2 and earlier versions which do not provide Extended Master Secret (EMS) support as outlined in RFC7627.
--

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is `INVALID_CLIENT`. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpConnectUrl** function is a high-level function that uses an FTP URL to establish a connection with a server. Unlike the other connection related functions such as **FtpConnect**, this function does more than simply connect to the server. It will also authenticate the client session, change the current working directory and set the default file transfer mode. By default, this function will always place the client in passive mode, ensuring the broadest compatibility with most servers.

The URL must be complete, and specify either a standard or secure FTP scheme:

```
[ftp|ftps|sftp]://[username : password] @[remotehost] [:remoteport] /  
[path / ...] [filename]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. If a path is specified as part of the URL, the function will attempt to change the current working directory. The paths in an FTP URL are relative to the home directory of the user account and are not absolute paths starting at the root directory on the server. If a file name is also specified in the URL, it will be ignored and only the file path will be used. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "ftp" scheme is used and the `FTP_OPTION_SECURE` option is specified, that option will be ignored. To establish a secure connection, either the "ftps" or "sftp" scheme must be specified.

The **FtpConnectUrl** function is designed to provide a simpler, more convenient interface to establishing a connection with a server. However, complex connections such as those using a proxy server or a secure connection which uses a client certificate will require the program to use the lower-level connection functions. If you only need to upload or download a file using a URL, then refer to the **FtpUploadFile** and **FtpDownloadFile** functions.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **FtpAttachThread** function.

Specifying the `FTP_OPTION_FREETHREAD` option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Example

```
HCLIENT hClient;  
LPCTSTR lpszURL = _T("ftp://ftp.sockettools.com/pub/");  
  
// Connect to the site specified by the URL
```

```
hClient = FtpConnectUrl(lpszURL, FTP_TIMEOUT, FTP_OPTION_DEFAULT);

if (hClient == INVALID_CLIENT)
{
    TCHAR szError[128];

    // Display a message box that describes the error
    FtpGetErrorString(FtpGetLastError(), szError, 128);
    MessageBox(NULL, szError, NULL, MB_ICONEXCLAMATION|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpConnect](#), [FtpDisconnect](#), [FtpDownloadFile](#), [FtpUploadFile](#), [FtpValidateUrl](#)

FtpCreateDirectory Function

```
INT WINAPI FtpCreateDirectory(  
    HCLIENT hClient,  
    LPCTSTR lpszDirectory  
);
```

The **FtpCreateDirectory** function creates the specified directory on the server.

Parameters

hClient

Handle to the client session.

lpszDirectory

Points to a string that specifies the name of the directory. The file pathing and name conventions must be that of the server.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function uses the MKD command to create the directory. The user must have the appropriate permission to create the specified directory.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpChangeDirectory](#), [FtpGetDirectory](#), [FtpRemoveDirectory](#)

FtpCreateSecurityCredentials Function

```
BOOL WINAPI FtpCreateSecurityCredentials(  
    DWORD dwProtocol,  
    DWORD dwOptions,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName,  
    LPVOID lpvReserved,  
    LPSECURITYCREDENTIALS* lppCredentials  
);
```

The **FtpCreateSecurityCredentials** function creates a **SECURITYCREDENTIALS** structure.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is

	supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.
SECURITY_PROTOCOL_SSH	Either version 1.0 or 2.0 of the Secure Shell protocol should be used when establishing the connection. The correct protocol is automatically selected based on the version of the protocol that is supported by the server.
SECURITY_PROTOCOL_SSH1	The Secure Shell 1.0 protocol should be used when establishing the connection. This is an older version of the protocol which should not be used unless explicitly required by the server. Most modern SSH server support version 2.0 of the protocol.
SECURITY_PROTOCOL_SSH2	The Secure Shell 2.0 protocol should be used when establishing the connection. This is the default version of the protocol that is supported by most SSH servers.

dwOptions

A value which specifies one or options. This value should always be zero for connections using SSH. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local

	machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A pointer to a string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A pointer to a string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A pointer to a string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

lpvReserved

Pointer reserved for future use. Set it to NULL when using this function.

lppCredentials

Pointer to an [LPSECURITYCREDENTIALS](#) pointer. The memory for the credentials structure will be allocated by this function and must be released by calling the **FtpDeleteSecurityCredentials** function when it is no longer needed. The pointer value must be set to NULL before the function is called. It is important to note that this is a pointer to a pointer variable, not a pointer to the SECURITYCREDENTIALS structure itself.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

The structure that is created by this function may be used as client credentials when establishing a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **FtpAsyncConnect** or **FtpConnect** functions.

Example

```
LPSECURITYCREDENTIALS lpSecCred = NULL;
FtpCreateSecurityCredentials(SEcurity_PROTOCOL_DEFAULT,
                             CREDENTIAL_STORE_CURRENT_USER,
                             NULL,
                             NULL,
                             strCertStore,
                             strCertName,
                             NULL,
                             &lpSecCred);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cstools10.h

Import Library: csWSKV10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpAsyncConnect](#), [FtpConnect](#), [FtpDeleteSecurityCredentials](#)

FtpDeleteFile Function

```
INT WINAPI FtpDeleteFile(  
    HCLIENT hClient,  
    LPCTSTR lpszFileName  
);
```

The **FtpDeleteFile** function deletes the specified file from the server.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file to delete. The file pathing and name conventions must be that of the server.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The current user must have the appropriate permission to delete the file, or an error will be returned by the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetFile](#), [FtpPutFile](#), [FtpRenameFile](#)

FtpDeleteSecurityCredentials Function

```
VOID WINAPI FtpDeleteSecurityCredentials(  
    LPSECURITYCREDENTIALS *LppCredentials  
);
```

The **FtpDeleteSecurityCredentials** function deletes an existing **SECURITYCREDENTIALS** structure.

Parameters

lppCredentials

Pointer to an **LPSECURITYCREDENTIALS** pointer. On exit from the function, the pointer will be NULL.

Return Value

None.

Example

```
if (lpSecCred)  
    FtpDeleteSecurityCredentials(&lpSecCred);  
FtpUninitialize();
```

Remarks

This function can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpCreateSecurityCredentials](#), [FtpUninitialize](#)

FtpDisableEvents Function

```
INT WINAPI FtpDisableEvents(  
    HCLIENT hClient  
);
```

The **FtpDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpDisableEvents** function is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this function has been called.

This function is automatically called if the client has event notification enabled, and the **FtpDisconnect** function is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

See Also

[FtpAsyncConnect](#), [FtpAsyncProxyConnect](#), [FtpEnableEvents](#), [FtpRegisterEvent](#)

FtpDisableTrace Function

```
BOOL WINAPI FtpDisableTrace();
```

The **FtpDisableTrace** function disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, a value of zero is returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[FtpEnableTrace](#)

FtpDisconnect Function

```
INT WINAPI FtpDisconnect(  
    HCLIENT hClient  
);
```

The **FtpDisconnect** function terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

See Also

[FtpConnect](#), [FtpProxyConnect](#), [FtpUninitialize](#)

FtpDownloadFile Function

```
BOOL WINAPI FtpDownloadFile(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszFileURL,  
    UINT nTimeout  
    DWORD dwOptions  
    LPFTPTRANSFERSTATUS lpStatus  
    FTPEVENTPROC lpEventProc  
    DWORD_PTR dwParam  
);
```

The **FtpDownloadFile** function downloads the specified file from the server to the local system.

Parameters

lpszLocalFile

A pointer to a string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszFileURL

A pointer to a string that specifies the complete URL of the file to be downloaded. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
FTP_OPTION_PASSIVE	This option specifies the client should attempt to establish the data connection with the server. When the client uploads or downloads a file, normally the server establishes a second connection back to the client which is used to transfer the file data. However, if the local system is behind a firewall or a NAT router, the server may not be able to create the data connection and the transfer will fail. By specifying this option, it forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data

	connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should use the AUTH command to negotiate an explicit secure connection. Some servers may only require this when connecting to the server on ports other than 990.

lpStatus

A pointer to an FTPTRANSFERSTATUS structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpDownloadFile** function provides a convenient way for an application to download a file in a single function call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session, change the current working directory if necessary and then download the file to the local system. The URL must be complete, and specify either a standard or secure FTP scheme:

```
[ftp|ftps|sftp]://[username : password] @[remotehost] [:remoteport] /
[path / ...] [filename] [;type=a|i]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. If a path is specified as part of the URL, the function will attempt to change the current working directory. Note that the path in an FTP URL is relative to the home directory of the user account and is not an absolute path starting at the root directory on the server. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "ftp" scheme is used and the FTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, either the "ftps" or "sftp" scheme must be specified.

The optional "type" value at the end of the file name determines if the file should be downloaded as a text or binary file. A value of "a" specifies that the file should be downloaded as a text file. A value of "i" specifies that the file should be downloaded as a binary file. If the type is not explicitly specified, the file will be downloaded as a binary file.

The *lpStatus* parameter can be used by the application to determine the final status of the transfer, including the total number of bytes copied, the amount of time elapsed and other

information related to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The *lpEventProc* parameter specifies a pointer to a function which will be periodically called during the file transfer process. This can be used to check the status of the transfer by calling **FtpGetTransferStatus** and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The *dwParam* parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the *this* pointer as the value, and then cast it back to an object pointer inside the callback function. If no event handler is required, then a NULL pointer can be specified as the value for *lpEventProc* and the *dwParam* parameter will be ignored.

The **FtpDownloadFile** function is designed to provide a simpler interface for downloading a file. However, complex connections such as those using a proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **FtpConnect** and then use **FtpGetFile** to download the file.

Example

```
FTPTRANSFERSTATUS ftpStatus;
LPCTSTR lpszLocalFile = _T("c:\\temp\\database.mdb");
LPCTSTR lpszFileURL = _T("ftp://ftp.example.com/updates/database.mdb");
BOOL bResult;

// Download the file using the specified URL
bResult = FtpDownloadFile(lpszLocalFile,
                          lpszFileURL,
                          FTP_OPTION_PASSIVE,
                          &ftpStatus,
                          NULL, 0);

if (!bResult)
{
    TCHAR szError[128];

    // Display a message box that describes the error
    FtpGetErrorString(FtpGetLastError(), szError, 128);
    MessageBox(NULL, szError, NULL, MB_ICONEXCLAMATION|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpEventProc](#), [FtpGetFile](#), [FtpGetTransferStatus](#), [FtpUploadFile](#), [FTPTRANSFERSTATUS](#)

FtpDownloadFileEx Function

```
BOOL WINAPI FtpDownloadFileEx(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszFileURL,  
    UINT nTimeout  
    DWORD dwOptions  
    LPFTPTRANSFERSTATUSEX lpStatus  
    FTPEVENTPROC lpEventProc  
    DWORD_PTR dwParam  
);
```

The **FtpDownloadFileEx** function downloads the specified file from the server to the local system. This version of the function is designed to support files that are larger than 4GB.

Parameters

lpszLocalFile

A pointer to a string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszFileURL

A pointer to a string that specifies the complete URL of the file to be downloaded. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
FTP_OPTION_PASSIVE	This option specifies the client should attempt to establish the data connection with the server. When the client uploads or downloads a file, normally the server establishes a second connection back to the client which is used to transfer the file data. However, if the local system is behind a firewall or a NAT router, the server may not be able to create the data connection and the transfer will fail. By specifying this option, it forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it

	returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should use the AUTH command to negotiate an explicit secure connection. Some servers may only require this when connecting to the server on ports other than 990.

lpStatus

A pointer to an FTPTRANSFERSTATUSEX structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpDownloadFileEx** function provides a convenient way for an application to download a file in a single function call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session, change the current working directory if necessary and then download the file to the local system. The URL must be complete, and specify either a standard or secure FTP scheme:

```
[ftp|ftps|sftp]://[username : password] @] remotehost [:remoteport] /
[path / ...] [filename] [;type=a|i]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. If a path is specified as part of the URL, the function will attempt to change the current working directory. Note that the path in an FTP URL is relative to the home directory of the user account and is not an absolute path starting at the root directory on the server. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "ftp" scheme is used and the FTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, either the "ftps" or "sftp" scheme must be specified.

The optional "type" value at the end of the file name determines if the file should be downloaded as a text or binary file. A value of "a" specifies that the file should be downloaded as a text file. A value of "i" specifies that the file should be downloaded as a binary file. If the type is not explicitly specified, the file will be downloaded as a binary file.

The *lpStatus* parameter can be used by the application to determine the final status of the

transfer, including the total number of bytes copied, the amount of time elapsed and other information related to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The *lpEventProc* parameter specifies a pointer to a function which will be periodically called during the file transfer process. This can be used to check the status of the transfer by calling **FtpGetTransferStatusEx** and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The *dwParam* parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the *this* pointer as the value, and then cast it back to an object pointer inside the callback function. If no event handler is required, then a NULL pointer can be specified as the value for *lpEventProc* and the *dwParam* parameter will be ignored.

The **FtpDownloadFileEx** function is designed to provide a simpler interface for downloading a file. However, complex connections such as those using a proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **FtpConnect** and then use **FtpGetFileEx** to download the file.

Example

```
FTPTRANSFERSTATUSEX ftpStatus;
LPCTSTR lpszLocalFile = _T("c:\\temp\\database.mdb");
LPCTSTR lpszFileURL = _T("ftp://ftp.example.com/updates/database.mdb");
BOOL bResult;

// Download the file using the specified URL
bResult = FtpDownloadFileEx(lpszLocalFile,
                           lpszFileURL,
                           FTP_OPTION_PASSIVE,
                           &ftpStatus,
                           NULL, 0);

if (!bResult)
{
    TCHAR szError[128];

    // Display a message box that describes the error
    FtpGetErrorString(FtpGetLastError(), szError, 128);
    MessageBox(NULL, szError, NULL, MB_ICONEXCLAMATION|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpEventProc](#), [FtpGetFileEx](#), [FtpGetTransferStatusEx](#), [FtpUploadFileEx](#), [FTPTRANSFERSTATUSEX](#)

FtpEnableEvents Function

```
INT WINAPI FtpEnableEvents(  
    HCLIENT hClient,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **FtpEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **FtpRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. This value must be greater than 1024.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpEnableEvents** function is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer.

The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Constant	Description
FTP_EVENT_CONNECT	The control connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
FTP_EVENT_DISCONNECT	The server has closed the control connection to the client. The client should read any remaining data and disconnect.
FTP_EVENT_OPENFILE	The data connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
FTP_EVENT_CLOSEFILE	The server has closed the data connection to the client. The client

	should read any remaining data and close the data channel.
FTP_EVENT_READFILE	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_WRITEFILE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
FTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
FTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
FTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.
FTP_EVENT_GETFILE	This event is generated when a file download has completed. If multiple files are being downloaded, this event will be generated for each file.
FTP_EVENT_PUTFILE	This event is generated when a file upload has completed. If multiple files are being uploaded, this event will be generated for each file.

It is not required that the client be placed in asynchronous mode in order to receive command and progress event notifications. To disable event notification, call the **FtpDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[FtpDisableEvents](#), [FtpRegisterEvent](#)

FtpEnableFeature Function

```
BOOL WINAPI FtpEnableFeature(  
    HCLIENT hClient,  
    DWORD dwFeature,  
    BOOL bEnable  
);
```

The **FtpEnableFeature** function enables or disables a specific server feature available to the client.

Parameters

hClient

Handle to the client session.

dwFeature

An unsigned integer which specifies the feature to be enabled to disabled for the current client session. Refer to the documentation for the **FtpGetFeatures** function for a list of available features.

bEnable

A boolean flag which specifies if the feature should be enabled or disabled. If the value is non-zero, the library will attempt to use that feature on the server. If the value is zero, the feature is disabled. If an application calls a function which requires a specific feature and that feature is disabled, the function will fail with an error indicating the feature is not supported.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it will return a value of zero. To get extended error information, call the **FtpGetLastError** function.

Remarks

The **FtpEnableFeature** function is used to enable or disable a specific feature for the current session. When a client connection is first established, features are enabled based on the server type and the server's response to the FEAT command. However, as the client issues commands to the server, if the server reports that the command is unrecognized that feature will automatically be disabled in the client. An application can use the **FtpEnableFeature** function to control what commands will be sent to the server, or re-enable a command that was previously disabled.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

See Also

[FtpGetFeatures](#), [FtpSetFeatures](#)

FtpEnableTrace Function

```
BOOL WINAPI FtpEnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **FtpEnableTrace** function enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace function logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpDisableTrace](#)

FtpEndOfFile Function

```
BOOL WINAPI FtpEndOfFile(  
    HCLIENT hClient  
);
```

The **FtpEndOfFile** function is used to determine if the end-of-file has been reached while reading the contents of a remote file.

Parameters

hClient

Handle to the client session.

Return Value

If the end-of-file has been reached, the function returns a non-zero value. If the client handle is invalid, or the end-of-file has not been reached, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

See Also

[FtpCloseFile](#), [FtpOpenFile](#), [FtpRead](#)

FtpEnumFiles Function

```
INT WINAPI FtpEnumFiles(  
    HCLIENT hClient,  
    LPCTSTR lpszDirectory,  
    LPCTSTR lpszFileMask,  
    DWORD dwOptions,  
    LPFTPFILESTATUS lpFileList,  
    INT nMaxFiles  
);
```

The **FtpEnumFilesEx** function populates an array of structures that contain information about the files in a directory.

Parameters

hClient

Handle to the client session.

lpszDirectory

A pointer to a string that specifies the name of a directory on the server. If this parameter is NULL or points to an empty string, the function will return the files in the current working directory. This string cannot contain wildcard characters and must specify a valid directory name that exists on the server.

lpszFileMask

A pointer to a string that specifies a wildcard file mask that is used to return a subset of files in the directory. If this parameter is NULL or an empty string then all of the files in the directory will be returned.

dwOptions

An unsigned integer value that specifies one or more options. This parameter can be a combination of one or more of the following values:

Constant	Description
FTP_ENUM_DEFAULT	The function will return both regular files and subdirectories.
FTP_ENUM_FILE	The function will return only regular files.
FTP_ENUM_DIRECTORY	The function will return only subdirectories.
FTP_ENUM_FULLPATH	The function will return the full path of the file or subdirectory.

lpFileList

A pointer to an array of [FTPFILESTATUS](#) structures which contains information about each of the files in the specified directory. This parameter cannot be NULL, and the array must be large enough to store the number of files specified by the *nMaxFiles* parameter.

nMaxFiles

An integer value that specifies the maximum number of files that should be returned. This value must be greater than zero and the *lpFileList* parameter must provide an array that is large enough to store information about each file.

Return Value

If the function succeeds, the return value is the number of files returned by the function. If the directory is empty or there are no files that match the specified wildcard file mask, the function will return zero. If the function fails, the return value is `FTP_ERROR`. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpEnumFiles** function provides a high-level interface for obtaining a list of available files in a directory on the server in a single function call. This is an alternative to opening a directory and returning information about each file by calling the **FtpGetNextFile** function in a loop.

This function temporarily changes the current working directory to the directory specified by the *lpSzDirectory* parameter. The current working directory will be restored to its original value when the function returns. The user must have the appropriate permissions to access the directory or this function will fail.

To obtain information on a subset of files in the directory, you can specify a wildcard file mask. For FTP and FTPS (SSL) sessions, this value is passed as a parameter to the LIST command and the server performs the wildcard matching. For SFTP (SSH) sessions the wildcard matching is performed by the library, and the standard conventions for Windows file wildcards are used.

By default, the *szFileName* member for each **FTPFILESTATUS** structure will contain the base file name. If the `FTP_ENUM_FULLPATH` option is specified, the function will return the full path name to the file. The library must be able to automatically determine the path delimiter that is used by the server. This is done by examining how the server identifies itself, the current directory format and the path the server returns for the current working directory. For example, UNIX based servers use the forward slash as a path delimiter. If the function cannot determine what the appropriate path delimiter is, it will ignore this option and return only the base file name.

This function will cause the current thread to block until the file listing completes, a timeout occurs or the operation is canceled. If the directory contains files that are larger than 4GB, the **FtpEnumFilesEx** function should be used to obtain the correct file size.

Example

```
LPFTPFILESTATUS lpFileList = new FTPFILESTATUS[MAXFILECOUNT];

// Return all of the regular files in the current working directory
INT nResult = FtpEnumFiles(hClient, NULL, NULL, FTP_ENUM_FILE,
                          lpFileList, MAXFILECOUNT);

if (nResult == FTP_ERROR)
{
    DWORD dwError = FtpGetLastError();
    _tprintf(_T("FtpEnumFiles failed, error 0x%08lx\n"), dwError);
    return;
}

_tprintf(_T("FtpEnumFiles returned %d files\n"), nResult);
for (INT nIndex = 0; nIndex < nResult; nIndex++)
{
    _tprintf(_T("file=\"%s\" size=%lu\n"), lpFileList[nIndex].szFileName,
            lpFileList[nIndex].dwFileSize);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpEnumFilesEx](#), [FtpGetFileStatus](#), [FtpGetFirstFile](#), [FtpGetNextFile](#), [FtpOpenDirectory](#)

FtpEnumFilesEx Function

```
INT WINAPI FtpEnumFilesEx(  
    HCLIENT hClient,  
    LPCTSTR lpszDirectory,  
    LPCTSTR lpszFileMask,  
    DWORD dwOptions,  
    LPFTPFILESTATUSEX lpFileList,  
    INT nMaxFiles  
);
```

The **FtpEnumFilesEx** function populates an array of structures that contain information about the files in a directory. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszDirectory

A pointer to a string that specifies the name of a directory on the server. If this parameter is NULL or points to an empty string, the function will return the files in the current working directory. This string cannot contain wildcard characters and must specify a valid directory name that exists on the server.

lpszFileMask

A pointer to a string that specifies a wildcard file mask that is used to return a subset of files in the directory. If this parameter is NULL or an empty string then all of the files in the directory will be returned.

dwOptions

An unsigned integer value that specifies one or more options. This parameter can be a combination of one or more of the following values:

Constant	Description
FTP_ENUM_DEFAULT	The function will return both regular files and subdirectories.
FTP_ENUM_FILE	The function will return only regular files.
FTP_ENUM_DIRECTORY	The function will return only subdirectories.
FTP_ENUM_FULLPATH	The function will return the full path of the file or subdirectory.

lpFileList

A pointer to an array of [FTPFILESTATUSEX](#) structures which contains information about each of the files in the specified directory. This parameter cannot be NULL, and the array must be large enough to store the number of files specified by the *nMaxFiles* parameter.

nMaxFiles

An integer value that specifies the maximum number of files that should be returned. This value must be greater than zero and the *lpFileList* parameter must provide an array that is large enough to store information about each file.

Return Value

If the function succeeds, the return value is the number of files returned by the function. If the directory is empty or there are no files that match the specified wildcard file mask, the function will return zero. If the function fails, the return value is `FTP_ERROR`. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpEnumFilesEx** function provides a high-level interface for obtaining a list of available files in a directory on the server in a single function call. This is an alternative to opening a directory and returning information about each file by calling the **FtpGetNextFileEx** function in a loop.

This function temporarily changes the current working directory to the directory specified by the *lpzDirectory* parameter. The current working directory will be restored to its original value when the function returns. The user must have the appropriate permissions to access the directory or this function will fail.

To obtain information on a subset of files in the directory, you can specify a wildcard file mask. For FTP and FTPS (SSL) sessions, this value is passed as a parameter to the LIST command and the server performs the wildcard matching. For SFTP (SSH) sessions the wildcard matching is performed by the library, and the standard conventions for Windows file wildcards are used.

By default, the *szFileName* member for each **FTPFILESTATUSEX** structure will contain the base file name. If the `FTP_ENUM_FULLPATH` option is specified, the function will return the full path name to the file. The library must be able to automatically determine the path delimiter that is used by the server. This is done by examining how the server identifies itself, the current directory format and the path the server returns for the current working directory. For example, UNIX based servers use the forward slash as a path delimiter. If the function cannot determine what the appropriate path delimiter is, it will ignore this option and return only the base file name.

This function will cause the current thread to block until the file listing completes, a timeout occurs or the operation is canceled.

Example

```
LPFTPFILESTATUSEX lpFileList = new FTPFILESTATUSEX[MAXFILECOUNT];

// Return all of the regular files in the current working directory
INT nResult = FtpEnumFilesEx(hClient, NULL, NULL, FTP_ENUM_FILE,
                             lpFileList, MAXFILECOUNT);

if (nResult == FTP_ERROR)
{
    DWORD dwError = FtpGetLastError();
    _tprintf(_T("FtpEnumFilesEx failed, error 0x%08lx\n"), dwError);
    return;
}

_tprintf(_T("FtpEnumFilesEx returned %d files\n"), nResult);
for (INT nIndex = 0; nIndex < nResult; nIndex++)
{
    _tprintf(_T("file=\"%s\" size=%I64d\n"), lpFileList[nIndex].szFileName,
            lpFileList[nIndex].uiFileSize);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpEnumFiles](#), [FtpGetFileStatusEx](#), [FtpGetFirstFileEx](#), [FtpGetNextFileEx](#), [FtpOpenDirectory](#)

FtpEnumTasks Function

```
INT WINAPI FtpEnumTasks(  
    UINT * lpTasks,  
    INT nMaxTasks,  
    DWORD dwOptions  
);
```

Return a list of active, suspended or finished asynchronous tasks.

Parameters

lpTasks

A pointer to an array of unsigned integer values that will contain unique task identifiers when the function returns. If this parameter is NULL, the function will return the number of tasks.

nMaxTasks

An integer value that specifies the maximum number of task identifiers that may be copied into the *lpTasks* array. If the *lpTasks* parameter is NULL, this value must be zero.

dwOptions

An unsigned integer that specifies the type of asynchronous tasks that may be returned by this function. It may be a combination of the following values:

Constant	Description
FTP_TASK_DEFAULT	The list of asynchronous task IDs should include both active and suspended tasks. This option is the same as specifying both the FTP_TASK_ACTIVE and FTP_TASK_SUSPENDED options.
FTP_TASK_ACTIVE	The list of asynchronous task IDs should include those tasks which are currently active. An active task represents a background connection to a server that is in the process of performing the requested action, such as uploading or downloading a file.
FTP_TASK_SUSPENDED	The list of asynchronous task IDs should include those tasks which have been suspended. A suspended task represents a background connection that has been established, but the worker thread is not scheduled for execution.
FTP_TASK_FINISHED	The list of asynchronous task IDs should include those tasks which have completed recently.

Return Value

If the function is successful, the return value is the number of task identifiers copied into the provided array. If there are no tasks which match the requested criteria, the return value is zero. A return value of FTP_ERROR indicates an error has occurred. To get extended error information, call the **FtpGetLastError** function.

Remarks

The **FtpEnumTasks** function can be used to obtain a list of numeric identifiers that represent the asynchronous tasks that have been started or those that have completed. These task IDs are used by other functions to reference the background worker thread that has been created and obtain

status information for the task. For example, the **FtpTaskDone** function can be used to determine if a particular task has completed, and the **FtpTaskWait** function can be used to wait for a task to complete and return an error status code if the background operation failed.

There is an internal limit of 128 asynchronous tasks per process that may be active at any one time. When a task completes, the status information about that task is maintained for period of time after the task has completed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[FtpTaskDone](#), [FtpTaskResume](#), [FtpTaskSuspend](#), [FtpTaskWait](#)

FtpEventProc Function

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

The **FtpEventProc** function is an application-defined callback function that processes events generated by the client.

Parameters

hClient

Handle to the client session.

nEventId

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **FtpRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **FtpRegisterEvent** function. This callback function is also used by asynchronous tasks to notify the application when the task has started and completed. The **FtpEventProc** function is a placeholder for the application-defined function name.

If the callback function is invoked by an asynchronous task, it will execute in the context of the worker thread that is managing the client session. You must ensure that any access to global or static variables are synchronized, otherwise the results may be unpredictable. It is recommended that you do not declare any static variables within the callback function itself.

If the application has a graphical user interface, you should never attempt to directly modify a UI control from within the callback function for an asynchronous task. Controls should only be modified by the same UI thread that created their window. One common approach to resolve this issue is to post a user-defined message to the main window to signal that the user interface needs to be updated. The message handler would then process the user-defined message and update the user interface as needed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

See Also

[FtpDisableEvents](#), [FtpEnableEvents](#), [FtpFreezeEvents](#), [FtpRegisterEvent](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

FtpFreezeEvents Function

```
INT WINAPI FtpFreezeEvents(  
    HCLIENT hClient,  
    BOOL bFreeze  
);
```

The **FtpFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpDisableEvents](#), [FtpEnableEvents](#), [FtpRegisterEvent](#)

FtpGetActivePorts Function

```
INT WINAPI FtpGetActivePorts(  
    HCLIENT hClient,  
    UINT * LpnLowPort,  
    UINT * LpnHighPort  
);
```

The **FtpGetActivePorts** function returns the local port numbers used for active mode file transfers.

Parameters

hClient

Handle to the client session.

lpnLowPort

Points to an unsigned integer that will contain the low port number when the function returns.

lpnHighPort

Points to an unsigned integer that will contain the high port number when the function returns.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function is used to determine the default port numbers being used for active mode file transfers. When using active mode, the client listens for an inbound connection from the server rather than establishing an outbound connection for the data transfer. In most cases, passive mode transfers are preferred because they mitigate potential compatibility issues with firewalls and NAT routers.

If active mode transfers are required, the default port range used when listening for the server connection is between 1024 and 5000. This is the standard range of ephemeral ports used by the Windows operating system. However, under some circumstances that range of ports may be too small, or a firewall may be configured to deny inbound connections on ephemeral ports. In that case, the **FtpSetActivePorts** function can be used to specify a different range of port numbers.

While it is technically permissible to assign the low and high port numbers to the same value, effectively specifying a single active port number, this is not recommended as it can cause the transfer to fail unexpectedly if multiple file transfers are performed. A minimum range of at least 1000 ports is recommended. For example, if you specify a low port value of 40000 then it is recommended that the high port value be at least 41000. The maximum port value is 65535.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpSetActivePorts](#), [FtpSetPassiveMode](#)

FtpGetBufferSize Function

```
INT WINAPI FtpGetBufferSize(  
    HCLIENT hClient  
);
```

The **FtpGetBufferSize** function returns the size in bytes of an internal buffer that will be used during data transfers.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the size of the internal buffer that will be used in data transfers. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The speed of data transfers, particularly on uploads, may be sensitive to network type and configuration, and the size of the internal buffer used for data transfers. The default size of this buffer will result in good performance for a wide range of network characteristics. A larger buffer will not necessarily result in better performance. For example, a value of 1460, which is the typical Maximum Transmission Unit (MTU), may be optimal in many situations.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

See Also

[FtpSetBufferSize](#)

FtpGetChannelMode Function

```
INT WINAPI FtpGetChannelMode(  
    HCLIENT hClient,  
    INT nChannel  
);
```

The **FtpGetChannelMode** function returns the mode of the specified communications channel.

Parameters

hClient

Handle to the client session.

nChannel

An integer value which specifies which channel to return information for. It may be one of the following values:

Constant	Description
FTP_CHANNEL_COMMAND	Return information about the command channel. This is the communication channel used to send commands to the server and receive command result and status information from the server.
FTP_CHANNEL_DATA	Return information about the data channel. This is the communication channel used to send or receive data during a file transfer.

Return Value

If the function succeeds, the return value is the mode for the specified channel. If the function fails, it will return FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetChannelMode** function returns a integer which may be one of the following values:

Constant	Description
FTP_CHANNEL_CLEAR	The channel is not encrypted. This is the default mode for both channels when a standard, non-secure connection is established with the server.
FTP_CHANNEL_SECURE	The channel is encrypted. This is the default mode for both channels when a secure connection is established with the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

See Also

[FtpSetChannelMode](#)

FtpGetClientQuota Function

```
INT WINAPI FtpGetClientQuota(  
    HCLIENT hClient,  
    LPFTPCLIENTQUOTA lpClientQuota  
);
```

The **GetClientQuota** function returns information about file quotas for the current client session.

Parameters

hClient

Handle to the client session.

lpClientQuota

A pointer to an [FTPCLIENTQUOTA](#) structure which contains the quota information returned by the server.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is `FTP_ERROR`. To get extended error information, call **GetLastError**.

Remarks

This method uses the XQUOTA command to obtain information for the current client session. If the server does not support this command, the function will fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

See Also

[FtpGetServerInformation](#), [FtpGetServerType](#)

FtpGetData Function

```
INT WINAPI FtpGetData(  
    HCLIENT hClient,  
    LPCTSTR lpszRemoteFile,  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwReserved  
);
```

The **FtpGetData** function transfers the contents of a file on the server to the specified buffer.

Parameters

hClient

Handle to the client session.

lpszRemoteFile

A pointer to a string that specifies the file on the server that will be transferred to the local system. The file naming conventions must be that of the host operating system.

lpvBuffer

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvBuffer* parameter. If the *lpvBuffer* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the file that was downloaded.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetData** function is used to download the contents of a remote file into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the *lpvBuffer* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvBuffer* parameter point to a global memory handle which will contain the file data when the function returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

The **FtpGetText** function can be used to download a text file and store the contents in a string.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the FTP_EVENT_PROGRESS event will be

periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **FtpEnableEvents**, or by registering a callback function using the **FtpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **FtpGetTransferStatus** function.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Return the file data into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter
nResult = FtpGetData(hClient,
                    lpszRemoteFile,
                    &hgblBuffer,
                    &cbBuffer,
                    0);

if (nResult != FTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // contents of the file data
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpChangeDirectory](#), [FtpEnableEvents](#), [FtpGetFile](#), [FtpGetText](#), [FtpGetTransferStatus](#), [FtpPutData](#), [FtpPutFile](#), [FtpRegisterEvent](#), [FtpSetBufferSize](#)

FtpGetDirectory Function

```
INT WINAPI FtpGetDirectory(  
    HCLIENT hClient,  
    LPTSTR lpszDirectory,  
    INT cbDirectory  
);
```

The **FtpGetDirectory** function copies the current working directory on the server to the specified buffer.

Parameters

hClient

Handle to the client session.

lpszDirectory

Points to a buffer that will contain the name of the current working directory on the server. The file pathing and name conventions must be that of the server.

cbDirectory

The maximum number of characters that may be copied into the buffer, including the terminating null-character.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function sends the PWD command to the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpChangeDirectory](#), [FtpCreateDirectory](#), [FtpRemoveDirectory](#)

FtpGetDirectoryFormat Function

```
INT WINAPI FtpGetDirectoryFormat(  
    HCLIENT hClient  
);
```

The **FtpGetDirectoryFormat** function returns an identifier which specifies what format is being used by the server to list files. By default, the library will automatically determine the appropriate format, but this value may be overridden by the **FtpSetDirectoryFormat** function.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the currently selected directory format. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

Refer to the **FtpSetDirectoryFormat** function for a list of directory format types that are supported by the library. This function can be used to determine which format was selected by the library after a file listing has been retrieved.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

See Also

[FtpCloseDirectory](#), [FtpGetFileStatus](#), [FtpGetFirstFile](#), [FtpGetNextFile](#), [FtpOpenDirectory](#), [FtpSetDirectoryFormat](#)

FtpGetErrorString Function

```
INT WINAPI FtpGetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);
```

The **FtpGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **FtpGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetLastError](#), [FtpSetLastError](#)

FtpGetFeatures Function

```
DWORD WINAPI FtpGetFeatures(  
    HCLIENT hClient  
);
```

The **FtpGetFeatures** function returns the server features available to the client.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is one or more bit flags which specify the features that are available to the client. If the function fails, it will return zero. Because it is possible that no features would be enabled, a return value of zero does not always indicate an error. An application should call **FtpGetLastError** to determine if an error code has been set.

Remarks

The **FtpGetFeatures** function returns a value which may be a combination of one or more of the following bit flags:

Constant	Description
FTP_FEATURE_SIZE	The server supports the SIZE command to determine the size of a file. If this feature is not enabled, the library will attempt to use the MLST or STAT command to determine the file size.
FTP_FEATURE_STAT	The server supports using the STAT command to return information about a specific file. If this feature is not enabled, the client may not be able to obtain information about a specific file such as its size, permissions or modification time.
FTP_FEATURE_MDTM	The server supports the MDTM command to obtain information about the modification time for a specific file. This command may also be used to set the file time on the server.
FTP_FEATURE_REST	The server supports restarting file transfers using the REST command. If this feature is not enabled, the client will not be able to restart file transfers and must upload or download the complete file.
FTP_FEATURE_SITE	The server supports site specific commands using the SITE command. If this feature is not enabled, no site specific commands will be sent to the server.
FTP_FEATURE_IDLE	The server supports setting the idle timeout period using the SITE IDLE command to specify the number of seconds that the client may idle before the server terminates the connection.
FTP_FEATURE_CHMOD	The server supports modifying the permissions of a specific file using the SITE CHMOD command. If this feature is not

	enabled, the client will not be able to set the permissions for a file.
FTP_FEATURE_AUTH	The server supports explicit SSL sessions using the AUTH command. If this feature is not enabled, the client will only be able to connect to a secure server that uses implicit SSL connections. Changing this feature has no effect on standard, non-secure connections.
FTP_FEATURE_PBSZ	The server supports the PBSZ command which specifies the buffer size used with secure data connections. If this feature is disabled, it may prevent the client from changing the protection level on the data channel. Changing this feature has no effect on standard, non-secure connections.
FTP_FEATURE_PROT	The server supports the PROT command which specifies the protection level for the data channel. If this feature is disabled, the client will be unable to change the protection level on the data channel. Changing this feature has no effect on standard, non-secure connections.
FTP_FEATURE_CCC	The server supports the CCC command which returns the command channel to a non-secure mode. Changing this feature has no effect on standard, non-secure connections.
FTP_FEATURE_HOST	The server supports the HOST command which enables a client to specify the hostname after establishing a connection with a server that supports virtual hosting.
FTP_FEATURE_MLST	The server supports the MLST command which returns status information for files. If this feature is enabled, the MLST command will be used instead of the STAT command.
FTP_FEATURE_MFMT	The server supports the MFMT command which is used to change the last modification time for a file. If this command is supported, it is used instead of the MDTM command to change the modification time for a file.
FTP_FEATURE_XCRC	The server supports the XCRC command which returns the CRC-32 checksum for the contents of a specified file. This command is used for file verification.
FTP_FEATURE_XMD5	The server supports the XMD5 command which returns an MD5 hash for the contents of a specified file. This command is used for file verification.
FTP_FEATURE_LANG	The server supports the LANG command which sets the language used for the current client session. Command responses and file naming conventions will use the specified language.
FTP_FEATURE_UTF8	The server supports the OPTS UTF-8 command which specifies UTF-8 encoding when specifying filenames. This feature is typically used in conjunction with setting the default language for the client session.
FTP_FEATURE_XQUOTA	The server supports the XQUOTA command which returns

	quota information for the current client session.
FTP_FEATURE_UTIME	The server supports the UTIME command which is used to change the last modification time for a specified file.

When a client connection is first established, features are enabled based on the server type and the server's response to the FEAT command. However, as the client issues commands to the server, if the server reports that the command is unrecognized that feature will automatically be disabled in the client.

For example, the first time an application calls the **FtpGetFileSize** function to determine the size of a file, the library will try to use the SIZE command. If the server reports that the SIZE command is not available, that feature will be disabled and the library will not use the command again during the session unless it is explicitly re-enabled. This is designed to prevent the library from repeatedly sending invalid commands to a server, which may result in the server aborting the connection.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

See Also

[FtpEnableFeature](#), [FtpSetFeatures](#)

FtpGetFile Function

```
INT WINAPI FtpGetFile(  
    HCLIENT hClient,  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszRemoteFile,  
    DWORD dwOptions,  
    DWORD dwOffset  
);
```

The **FtpGetFile** function transfers the specified file on the server to the local system.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A pointer to a string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A pointer to a string that specifies the file on the server that will be transferred to the local system. The file naming conventions must be that of the host operating system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Constant	Description
FTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the local file exists, it will be overwritten with the contents of the downloaded file.
FTP_TRANSFER_APPEND	This option specifies that if the local file exists, the contents of file on the server is appended to the local file. If the local file does not exist, it is created.

dwOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **FtpEnableEvents**, or by registering a callback function using the **FtpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **FtpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpChangeDirectory](#), [FtpEnableEvents](#), [FtpGetMultipleFiles](#), [FtpGetTransferStatus](#), [FtpPutFile](#), [FtpPutMultipleFiles](#), [FtpRegisterEvent](#), [FtpSetBufferSize](#), [FtpVerifyFile](#)

FtpGetFileEx Function

```
INT WINAPI FtpGetFileEx(  
    HCLIENT hClient,  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszRemoteFile,  
    DWORD dwOptions,  
    ULARGE_INTEGER uiOffset  
);
```

The **FtpGetFileEx** function transfers the specified file on the server to the local system. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A pointer to a string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A pointer to a string that specifies the file on the server that will be transferred to the local system. The file naming conventions must be that of the host operating system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Constant	Description
FTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the local file exists, it will be overwritten with the contents of the downloaded file.
FTP_TRANSFER_APPEND	This option specifies that if the local file exists, the contents of file on the server is appended to the local file. If the local file does not exist, it is created.

uiOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **FtpEnableEvents**, or by registering a callback function using

the **FtpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **FtpGetTransferStatusEx** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpChangeDirectory](#), [FtpEnableEvents](#), [FtpGetMultipleFiles](#), [FtpGetTransferStatusEx](#), [FtpPutFile](#), [FtpPutMultipleFiles](#), [FtpRegisterEvent](#), [FtpSetBufferSize](#), [FtpVerifyFile](#)

FtpGetFileList Function

```
INT WINAPI FtpGetFileList(  
    HCLIENT hClient,  
    LPCTSTR lpszDirectory,  
    DWORD dwOptions,  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);
```

The **FtpGetFileList** function returns an unparsed list of files in the specified directory.

Parameters

hClient

Handle to the client session.

lpszDirectory

A pointer to a string that specifies the name of a directory and/or a wildcard file mask. The format of the directory name must match the file naming conventions of the server. If this parameter is NULL or points to an empty string, the current working directory will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Constant	Description
FTP_LIST_DEFAULT	This option specifies the server should return a complete listing of files in the specified directory with as much detail as possible. This typically means that the file size, date, ownership and access rights will be returned to the client. Information about the files are returned in lines of text, with each line terminated by carriage return and linefeed (CRLF) characters. The exact format of the data returned is specific to the server operating system.
FTP_LIST_NAMEONLY	This option specifies the server should only return a list of file names, with no additional information about the file. Each file name is terminated by carriage return and linefeed (CRLF) characters.

lpszBuffer

A pointer to a string buffer that will contain the list of files when the function returns. This buffer should be large enough to store the complete file listing and a terminating null character. If the buffer is smaller than the total amount of data returned by the server, the data will be truncated. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the number of bytes copied into the string buffer, not including the terminating null character. If the function fails, the return value is FTP_ERROR. To get

extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetFileList** function returns a list of files in the specified directory, copying the data to a string buffer. Unlike the other functions like **FtpEnumFiles** that parse a directory listing and return information in an **FTPFILESTATUS** structure, this function returns the unparsed file list data. The actual format of the data that is returned depends on the operating system and how the server implements file listings. For example, UNIX servers typically return the output from the **/bin/lis** command.

Some servers may not support file listings for any directory other than the current working directory. If an error is returned when specifying a directory name, try changing the current working directory using the **FtpChangeDirectory** function and then call this function again, passing NULL or an empty string as the *lpszDirectory* parameter.

This function can be particularly useful when the client is connected to a server that returns file listings in a format that is not recognized by the library. The application can retrieve the unparsed file listing from the server and parse the contents. Note that if you specify the **FTP_LIST_NAMEONLY** option, the data will only contain a list of file names and there will be no way for the application to know if they represent a regular file or a subdirectory.

This function is supported for both FTP and SFTP (SSH) connections, however the format of the data may differ depending on which protocol is used. Most UNIX based FTP servers will not list files and subdirectories that begin with a period, however most SFTP servers will return a list of all files, even those that begin with a period.

This function will cause the current thread to block until the file listing completes, a timeout occurs or the operation is canceled.

Example

```
TCHAR szFileList[MAXFILELISTSIZE];

nResult = FtpGetFileList(hClient, NULL, FTP_LIST_NAMEONLY,
                        szFileList, MAXFILELISTSIZE);

if (nResult != FTP_ERROR)
    _tprintf(_T("%s\n"), szFileList);
else
{
    DWORD dwError = FtpGetLastError();
    _tprintf(_T("Unable to list files (error 0x%08lx)\n"), dwError);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpCloseDirectory](#), [FtpEnumFiles](#), [FtpGetDirectoryFormat](#), [FtpGetFirstFile](#), [FtpOpenDirectory](#), [FtpGetNextFile](#)

FtpGetFileNameEncoding Function

```
INT WINAPI FtpGetFileNameEncoding(  
    HCLIENT hClient  
);
```

The **FtpGetFileNameEncoding** function returns an identifier which specifies what type of encoding is being used when file names are sent to the server.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the type of encoding that is used. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

Refer to the **FtpSetFileNameEncoding** function for a list of encoding types that are supported by the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

See Also

[FtpCommand](#), [FtpEnableFeature](#) [FtpGetFeatures](#), [FtpSetFileNameEncoding](#), [FtpSetFeatures](#)

FtpGetFilePermissions Function

```
INT WINAPI FtpGetFilePermissions(  
    HCLIENT hClient,  
    LPCTSTR lpszFileName,  
    LPDWORD lpdwPermissions  
);
```

The **FtpGetFilePermissions** function returns information about the access permissions for a specific file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

A pointer to a string which contains the name of the file that the access permissions are to be returned for. The filename cannot contain any wildcard characters.

lpdwPermissions

A pointer to an unsigned integer which will contain the access permissions for the file when the function returns. The file permissions are represented as bit flags, and may be one or more of the following values:

Constant	Description
FILE_OWNER_READ	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.
FILE_OWNER_WRITE	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_OWNER_EXECUTE	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_GROUP_READ	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
FILE_GROUP_WRITE	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_GROUP_EXECUTE	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and

	search for files in that directory.
FILE_WORLD_READ	All users have permission to open the file for reading. This permission grants any user the right to download the file to the local system.
FILE_WORLD_WRITE	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.
FILE_WORLD_EXECUTE	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.

Return Value

If the function succeeds, the return value is a result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function uses the STAT command to retrieve information about the specified file. On some systems, the STAT command will not return information on files that contain spaces or tabs in the filename. In this case, the function will fail and value pointed to by the *lpdwPermissions* parameter will be zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetFileStatus](#), [FtpSetFilePermissions](#)

FtpGetFileSize Function

```
INT WINAPI FtpGetFileSize(  
    HCLIENT hClient,  
    LPCTSTR lpszFileName,  
    LPDWORD lpdwFileSize  
);
```

The **FtpGetFileSize** function returns the size of the specified file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file.

lpdwFileSize

Points to an unsigned integer that will contain the size of the specified file in bytes.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function uses the SIZE command to determine the length of the specified file. Not all servers implement this command, in which case the function call will fail, and the *lpdwFileSize* parameter will be set to zero.

Note that if the file on the server is a text file, it is possible that the value returned by this method will not match the size of the file when it is downloaded to the local system. This is because different operating systems use different sequences of characters to mark the end of a line of text, and when a file is transferred in text mode, the end of line character sequence is automatically converted to a carriage return-linefeed, which is the convention used by the Windows platform.

Some FTP servers will refuse to return the size of a file if the current file type is set to FILE_TYPE_ASCII because the size of a text file on the server may not accurately reflect what the size of the file will be on the local system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetFileStatus](#), [FtpOpenDirectory](#)

FtpGetFileSize Function

```
INT WINAPI FtpGetFileSizeEx(  
    HCLIENT hClient,  
    LPCTSTR lpszFileName,  
    ULARGE_INTEGER* lpuiFileSize  
);
```

The **FtpGetFileSizeEx** function returns the size of the specified file on the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file.

lpuiFileSize

Points to a ULARGE_INTEGER structure that will contain the size of the specified file in bytes.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function uses the SIZE command to determine the length of the specified file. Not all servers implement this command, in which case the function call will fail, and the *lpdwFileSize* parameter will be set to zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetFileStatusEx](#), [FtpOpenDirectory](#)

FtpGetFileStatus Function

```
BOOL WINAPI FtpGetFileStatus(  
    HCLIENT hClient,  
    LPCTSTR lpszFileName,  
    LPFTPFILESTATUS lpFileStatus  
);
```

The **FtpGetFileStatus** function returns information about a specific file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

A pointer to a string which contains the name of the file that status information will be returned on. The file name cannot contain any wildcard characters.

lpFileStatus

A pointer to an [FTPFILESTATUS](#) structure which contains information about the file returned by the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

This function uses the STAT command to retrieve information about the specified file. Unlike the **FtpGetFirstFile** and **FtpGetNextFile** functions, which read through a file list returned on the data channel, this function reads the result of a command string. For applications that need information about a specific file, using this function can be considerably faster than iterating through all of the files in a given directory. Note that not all servers support using the command in this way.

On some systems, the STAT command will not return information on files that contain spaces or tabs in the filename. In this case, the FTPFILESTATUS structure members will be empty strings and zero values.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpCloseDirectory](#), [FtpGetDirectoryFormat](#), [FtpGetFirstFile](#), [FtpGetNextFile](#), [FtpGetTransferStatus](#), [FtpOpenDirectory](#), [FtpSetDirectoryFormat](#)

FtpGetFileStatusEx Function

```
BOOL WINAPI FtpGetFileStatusEx(  
    HCLIENT hClient,  
    LPCTSTR lpszFileName,  
    LPFTPFILESTATUSEX lpFileStatus  
);
```

The **FtpGetFileStatusEx** function returns information about a specific file on the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszFileName

A pointer to a string which contains the name of the file that status information will be returned on. The file name cannot contain any wildcard characters.

lpFileStatus

A pointer to an [FTPFILESTATUSEX](#) structure which contains information about the file returned by the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

This function uses the STAT command to retrieve information about the specified file. Unlike the **FtpGetFirstFileEx** and **FtpGetNextFileEx** functions, which read through a file list returned on the data channel, this function reads the result of a command string. For applications that need information about a specific file, using this function can be considerably faster than iterating through all of the files in a given directory. Note that not all servers support using the command in this way.

On some systems, the STAT command will not return information on files that contain spaces or tabs in the filename. In this case, the FTPFILESTATUSEX structure members will be empty strings and zero values.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpCloseDirectory](#), [FtpGetDirectoryFormat](#), [FtpGetFirstFileEx](#), [FtpGetNextFileEx](#), [FtpGetTransferStatusEx](#), [FtpOpenDirectory](#), [FtpSetDirectoryFormat](#)

FtpGetFileTime Function

```
INT WINAPI FtpGetFileTime(  
    HCLIENT hClient,  
    LPCTSTR lpszFileName,  
    LPSYSTEMTIME lpFileTime,  
    BOOL bLocalize  
);
```

The **FtpGetFileTime** function returns the modification time for the specified file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file.

lpFileTime

Points to a [SYSTEMTIME](#) structure that will be set to the current modification time for the remote file.

bLocalize

A boolean flag which specifies if the file time is localized to the current timezone. If this value is non-zero, then the file time is adjusted to that the time is local to the current system. If this value is zero, the file time is returned in UTC time.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is `FTP_ERROR`. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetFileTime** function can be used to determine the date and time that a file was last modified on the server. The time may either be localized to the current system, or it may be returned as UTC time. If you plan on changing the values returned in the `SYSTEMTIME` structure and then calling **FtpSetFileTime** function to modify the file time on the server, you should do not localize the time.

This function uses the `MDTM` command to determine the modification time of the specified file. If the server does not support this command, the function will attempt to use the `STAT` command to determine the file modification time.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetFileStatus](#), [FtpOpenDirectory](#), [FtpSetFileTime](#)

FtpGetFileType Function

```
INT WINAPI FtpGetFileType(  
    HCLIENT hClient,  
    UINT * lpnFileType  
);
```

The **FtpGetFileType** function returns the default file type for the current client session.

Parameters

hClient

Handle to the client session.

lpnFileType

A pointer to an unsigned integer that will identify the current file type. This parameter cannot be NULL. When the function returns it will contain one of the following values:

Value	Description
FILE_TYPE_AUTO	The file type should be automatically determined based on the file name extension. If the file extension is unknown, the file type should be determined based on the contents of the file. The library has an internal list of common text file extensions, and additional file extensions can be registered using the FtpRegisterFileType function.
FILE_TYPE_ASCII	The file is a text file using the ASCII character set. For those servers which mark the end of a line with characters other than a carriage return and linefeed, it will be converted to the native client format. This is the file type used for directory listings.
FILE_TYPE_EBCDIC	The file is a text file using the EBCDIC character set. Local files will be converted to EBCDIC when sent to the server. Remote files will be converted to the native ASCII character set when retrieved from the server.
FILE_TYPE_IMAGE	The file is a binary file and no data conversion of any type is performed on the file. This is the default file type for most data files and executables. If the type of file cannot be automatically determined, it will always be considered a binary file.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

If this function is called when connected to an SFTP (SSH) server, the default file type will always be FILE_TYPE_IMAGE because SFTP does not differentiate between text files and binary files.

The **FtpSetFileType** function can be used to change the default file type.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[FtpOpenFile](#), [FtpRegisterFileType](#), [FtpSetFileMode](#), [FtpSetFileStructure](#), [FtpSetFileType](#),
[FtpSetPassiveMode](#)

FtpGetFirstFile Function

```
BOOL WINAPI FtpGetFirstFile(  
    HCLIENT hClient,  
    LPFTPFILESTATUS lpFileStatus  
);
```

The **FtpGetFirstFile** function returns the first file in the directory listing returned by the server after a call to the **FtpOpenDirectory** function.

Parameters

hClient

Handle to the client session.

lpFileStatus

A pointer to an [FTPFILESTATUS](#) structure which contains information about the file returned by the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

This file list information returned by the server is cached by the library, allowing you to use this function to reposition back to the beginning of the file list.

Example

```
if (FtpOpenDirectory(hClient, _T("")) != FTP_ERROR)  
{  
    FTPFILESTATUS ftpFile;  
    BOOL bResult;  
  
    bResult = FtpGetFirstFile(hClient, &ftpFile);  
    while (bResult)  
    {  
        // The ftpFile structure contains information about the file  
        bResult = FtpGetNextFile(hClient, &ftpFile);  
    }  
  
    FtpCloseDirectory(hClient);  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpCloseDirectory](#), [FtpGetDirectoryFormat](#), [FtpGetFileStatus](#), [FtpGetNextFile](#), [FtpOpenDirectory](#), [FtpSetDirectoryFormat](#)

FtpGetFirstFileEx Function

```
BOOL WINAPI FtpGetFirstFileEx(  
    HCLIENT hClient,  
    LPFTPFILESTATUSEX lpFileStatus  
);
```

The **FtpGetFirstFileEx** function returns the first file in the directory listing returned by the server after a call to the **FtpOpenDirectory** function. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpFileStatus

A pointer to an [FTPFILESTATUSEX](#) structure which contains information about the file returned by the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

This file list information returned by the server is cached by the library, allowing you to use this function to reposition back to the beginning of the file list.

Example

```
if (FtpOpenDirectory(hClient, _T("")) != FTP_ERROR)  
{  
    FTPFILESTATUSEX ftpFile;  
    BOOL bResult;  
  
    bResult = FtpGetFirstFileEx(hClient, &ftpFile);  
    while (bResult)  
    {  
        // The ftpFile structure contains information about the file  
        bResult = FtpGetNextFileEx(hClient, &ftpFile);  
    }  
  
    FtpCloseDirectory(hClient);  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpCloseDirectory](#), [FtpGetDirectoryFormat](#), [FtpGetFileStatusEx](#), [FtpGetNextFileEx](#), [FtpOpenDirectory](#), [FtpSetDirectoryFormat](#)

FtpGetLastError Function

DWORD WINAPI FtpGetLastError();

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **FtpSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **FtpGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **FtpSetLastError(0)** when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or FTP_ERROR. Those functions which call **FtpSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[FtpGetErrorString](#), [FtpSetLastError](#)

FtpGetMultipleFiles Function

```
INT WINAPI FtpGetMultipleFiles(  
    HCLIENT hClient,  
    LPCTSTR lpszLocalDirectory,  
    LPCTSTR lpszRemoteDirectory,  
    LPCTSTR lpszFileMask,  
    DWORD dwReserved  
);
```

The **FtpGetMultipleFiles** function copies one or more files from the server to the local host, using the specified wildcard.

Parameters

hClient

Handle to the client session.

lpszLocalDirectory

Pointer to a string which specifies the local directory where the files will be copied to. A NULL pointer or empty string specifies that files should be copied to the current working directory.

lpszRemoteDirectory

Pointer to a string which specifies the remote directory where the files will be copied from. A NULL pointer or empty string specifies that the files should be copied from the current working directory on the server.

lpszFileMask

Pointer to a string which specifies the files that are to be copied from the server to the local system. The file mask should follow the native conventions used for wildcard file matches on the server.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetMultipleFiles** function is used to transfer files from the server to the local host which match a specified wildcard file mask. This function requires that the client be able to automatically list and parse directory listings from the server, otherwise an error will be returned. All files will be transferred using the current file type as specified by the **FtpSetFileType** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpChangeDirectory](#), [FtpGetFile](#), [FtpPutFile](#), [FtpPutMultipleFiles](#), [FtpSetFileType](#)

FtpGetNextFile Function

```
BOOL WINAPI FtpGetNextFile(  
    HCLIENT hClient,  
    LPFTPFILESTATUS lpFileStatus  
);
```

The **FtpGetNextFile** function returns the next file in the directory listing returned by the server.

Parameters

hClient

Handle to the client session.

lpFileStatus

A pointer to an [FTPFILESTATUS](#) structure which contains information about the file returned by the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetNextFile** function returns the next file in the directory listing. If the last file has been returned, the function will return zero and the client should call the **FtpCloseDirectory** function to close the directory.

Example

```
if (FtpOpenDirectory(hClient, _T("")) != FTP_ERROR)  
{  
    FTPFILESTATUS ftpFile;  
    BOOL bResult;  
  
    bResult = FtpGetFirstFile(hClient, &ftpFile);  
    while (bResult)  
    {  
        // The ftpFile structure contains information about the file  
        bResult = FtpGetNextFile(hClient, &ftpFile);  
    }  
  
    FtpCloseDirectory(hClient);  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpCloseDirectory](#), [FtpGetDirectoryFormat](#), [FtpGetFileStatus](#), [FtpGetFirstFile](#), [FtpOpenDirectory](#), [FtpSetDirectoryFormat](#)

FtpGetNextFileEx Function

```
BOOL WINAPI FtpGetNextFile(  
    HCLIENT hClient,  
    LPFTPFILESTATUSEX lpFileStatus  
);
```

The **FtpGetNextFileEx** function returns the next file in the directory listing returned by the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpFileStatus

A pointer to an [FTPFILESTATUSEX](#) structure which contains information about the file returned by the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetNextFileEx** function returns the next file in the directory listing. If the last file has been returned, the function will return zero and the client should call the **FtpCloseDirectory** function to close the directory.

Example

```
if (FtpOpenDirectory(hClient, _T("")) != FTP_ERROR)  
{  
    FTPFILESTATUSEX ftpFile;  
    BOOL bResult;  
  
    bResult = FtpGetFirstFileEx(hClient, &ftpFile);  
    while (bResult)  
    {  
        // The ftpFile structure contains information about the file  
        bResult = FtpGetNextFileEx(hClient, &ftpFile);  
    }  
  
    FtpCloseDirectory(hClient);  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpCloseDirectory](#), [FtpGetDirectoryFormat](#), [FtpGetFileStatusEx](#), [FtpGetFirstFileEx](#), [FtpOpenDirectory](#), [FtpSetDirectoryFormat](#)

FtpGetPriority Function

```
INT WINAPI FtpGetPriority(  
    HCLIENT hClient  
);
```

The **FtpGetPriority** function returns a value which specifies the priority of file transfers.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the current file transfer priority. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetPriority** function can be used to determine the current priority assigned to file transfers performed by the client. It may be one of the following values:

Constant	Description
FTP_PRIORITY_NORMAL	The default priority which balances resource utilization and transfer speed. It is recommended that most applications use this priority.
FTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the transfer. It is typically used with worker threads running in the background when the amount of time required perform the transfer is not critical.
FTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the transfer and meters the bandwidth allocated for the transfer. This priority will increase the average amount of time required to complete a file transfer.
FTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the transfer, allocating more memory for internal buffering. It can be used when it is important to transfer the file quickly, and there are no other threads currently performing file transfers at the time.
FTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization while attempting to transfer the file as quickly as possible. If the file transfer is being performed in the main UI thread, this priority can cause the application to appear to become non-responsive. No events will be generated during the transfer.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpSetPriority](#)

FtpGetProxyType Function

```
INT WINAPI FtpGetProxyType(  
    HCLIENT hClient  
);
```

The **FtpGetProxyType** function returns the type of proxy that the client is connected to. By default, no proxy server is specified and this function returns a value of FTP_PROXY_NONE. For a list of possible proxy server types, refer to the **FtpProxyConnect** function.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value identifies the type of proxy that the client is connected to. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpConnect](#), [FtpProxyConnect](#)

FtpGetResultCode Function

```
INT WINAPI FtpGetResultCode(  
    HCLIENT hClient  
);
```

The **FtpGetResultCode** function reads the result code returned by the server in response to a command. The result code is a three-digit numeric code, and indicates if the operation succeeded, failed or requires additional action by the client.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

Result codes are three-digit numeric values returned by the server. They may be broken down into the following ranges:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

See Also

[FtpCommand](#), [FtpGetResultString](#)

FtpGetResultString Function

```
INT WINAPI FtpGetResultString(  
    HCLIENT hClient,  
    LPTSTR lpszResult,  
    INT cbResult  
);
```

The **FtpGetResultString** function returns the last message sent by the server along with the result code.

Parameters

hClient

Handle to the client session.

lpszResult

A pointer to the buffer that will contain the result string returned by the server.

cbResult

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server.

If the function fails, the return value is `FTP_ERROR`. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetResultString** function is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **FtpGetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpCommand](#), [FtpGetResultCode](#)

FtpGetSecurityInformation Function

```
BOOL WINAPI FtpGetSecurityInformation(  
    HCLIENT hClient,  
    LPSECURITYINFO lpSecurityInfo  
);
```

The **FtpGetSecurityInformation** function returns security protocol, encryption and certificate information about the current client connection.

Parameters

hClient

Handle to the client session.

lpSecurityInfo

A pointer to a [SECURITYINFO](#) structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

This function is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value `SECURITY_PROTOCOL_NONE`.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;  
  
securityInfo.dwSize = sizeof(SECURITYINFO);  
if (FtpGetSecurityInformation(hClient, &securityInfo))  
{  
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)  
    {  
        MessageBox(NULL, _T("The connection is not secure"),  
                    _T("Connection"), MB_OK);  
    }  
    else  
    {  
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)  
        {  
            MessageBox(NULL, _T("The connection is secure"),  
                        _T("Connection"), MB_OK);  
        }  
        else  
        {  
            MessageBox(NULL, _T("The server certificate not valid"),  
                        _T("Connection"), MB_OK);  
        }  
    }  
}
```

```
}  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpAsyncConnect](#), [FtpAsyncProxyConnect](#), [FtpConnect](#), [FtpCreateSecurityCredentials](#),
[FtpDeleteSecurityCredentials](#), [FtpDisconnect](#), [FtpProxyConnect](#), [SECURITYINFO](#)

FtpGetServerInformation Function

```
INT WINAPI FtpGetServerInformation(  
    HCLIENT hClient,  
    LPTSTR LpszSystemInfo,  
    INT nMaxLength  
);
```

The **FtpGetServerInformation** function returns information about the server, typically including the operating system type, version and platform.

Parameters

hClient

Handle to the client session.

lpszSystemInfo

A pointer to the buffer that will contain the system information returned by the server.

nMaxLength

The maximum number of characters that can be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function sends the SYST command to the server. The first word will identify the type of operating system. The format for the remaining information depends on the server type. Typically it is a description of the operating system version and hardware platform. For example, here are some strings commonly returned by various FTP servers:

Example	Description
UNIX Type: L8	A standard UNIX based server. This is the most common value returned by servers, and this indicates that the server supports UNIX file naming and directory listing conventions. This string may also include additional information such as the specific variant of UNIX and its version. The L8 portion of the string is a convention that lets the client know that a byte consists of 8 bits.
Windows_NT Version 5.0	A standard Windows based server, typically part of Internet Information Services (ISS). The server will use Windows file naming and directory listing conventions. The version identifies the specific release of Windows. For example, version 4.0 specifies Windows NT 4.0 and 5.0 specifies Windows 2000.
VMS V7.1 AlphaServer	A server running the VMS operating system. The server will use the standard file naming and directory listing conventions for that platform. Note that it is possible that a VMS system may also be configured to operate in a UNIX emulation mode, in which case it will return UNIX instead of VMS.
NetWare	A server running the NetWare operating system. The server will use

system type	the standard file naming and directory listing conventions for that platform. Note that it is possible that a NetWare system may be configured to operate in a UNIX emulation mode, in which case it return UNIX instead of NetWare.
WORLDGROUP Type: L8	A server running the WorldGroup software on the Windows platform. This server supports UNIX file naming and directory listing conventions. WorldGroup is a collaborative workgroup, email and file sharing service which includes an FTP server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpConnect](#), [FtpGetDirectoryFormat](#), [FtpGetClientQuota](#), [FtpGetServerStatus](#),
[FtpGetServerTimeZone](#), [FtpGetServerType](#)

FtpGetServerStatus Function

```
INT WINAPI FtpGetServerStatus(  
    HCLIENT hClient,  
    LPTSTR lpszStatus,  
    INT nMaxLength  
);
```

The **FtpGetServerStatus** function requests that the server return status information about itself.

Parameters

hClient

Handle to the client session.

lpszStatus

A pointer to the buffer that will contain the system status returned by the server.

nMaxLength

The maximum number of characters that can be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function sends the STAT command to the server. The format for the information returned depends on the server type. Typically it is a description of the server platform, version, current user and file transfer options.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetServerInformation](#), [FtpGetServerType](#)

FtpGetServerTimeZone Function

```
INT WINAPI FtpGetServerTimeZone(  
    HCLIENT hClient,  
    LPLONG lpnTimeZone  
);
```

The **FtpGetServerTimeZone** function returns the timezone for the current server.

Parameters

hClient

Handle to the client session.

lpnTimeZone

A pointer to a signed long integer which will contain the timezone offset in seconds.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function sends the SITE ZONE command to the server to determine its timezone. The value returned is expressed as the number of seconds offset from Coordinated Universal Time (UTC). A positive value specifies a time west of UTC, while a negative value specifies a time east of UTC. For example, a value of 28800 would specify an offset of 8 hours west of UTC, which is the Pacific timezone.

The SIZE ZONE command is an extension that is not supported by all servers. If the server timezone cannot be determined, the function will fail and the value pointed to by the *lpnTimeZone* parameter will be zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

See Also

[FtpGetClientQuota](#), [FtpGetServerInformation](#), [FtpGetServerStatus](#)

FtpGetServerType Function

```
INT WINAPI FtpGetServerType(  
    HCLIENT hClient  
);
```

The **FtpGetServerType** function returns the type of server the client has connected to.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is a numeric value which indicates the server type. If the function fails, the return value is FTP_ERROR. To get extended error information, call

FtpGetLastError.

Remarks

This function sends the SYST command to the server to determine the server type. The following server types are recognized by the library:

Constant	Description
FTP_SERVER_UNKNOWN	The server type could not be determined by issuing the SYST command. The server may not support the command, or the command may only be allowed when issued by an authenticated user.
FTP_SERVER_UNIX	The server is running on a UNIX based operating system. This can include Linux and other variants, as well as operating systems which emulate UNIX style file pathing and directory listings.
FTP_SERVER_MSDOS	The server is running on an MS-DOS based operating system. The server expects file pathing and naming conventions according to the standard MS-DOS format and returns directory listings similar to the output of the DIR command.
FTP_SERVER_WINDOWS	The server is running on a Windows based operating system. The server expects file pathing and naming conventions according to the standard Windows long filename format, and returns directory listings similar to the output of the DIR command. Note that Windows servers may be configured to return file and directory information in a format similar to UNIX systems, in which case the system may be identified as UNIX even though it is actually running on a Windows platform.
FTP_SERVER_VMS	The server is running on a DEC VMS based operating system. The server expects file pathing and naming conventions specific to that operating system. Note that VMS servers may be configured to return file and directory information in a format similar to UNIX systems, in which

	case the system may be identified as UNIX even though it is actually running on a VMS platform.
FTP_SERVER_NETWARE	The server is running on a NetWare based operating system. The server expects file pathing and naming conventions similar to the standard Windows long filename format, and returns directory listings that are similar to UNIX systems with the exception of the access and permissions flags for the file. Note that a NetWare system may return listings in different formats based on the filesystem and site specific options specified.
FTP_SERVER_OTHER	The server type was not recognized. An attempt will be made to automatically determine the correct file pathing and naming conventions used by the server. To obtain a list of files on the server, it may be necessary to use the FtpSetDirectoryFormat function to specify the directory listing format.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[FtpGetClientQuota](#), [FtpGetServerInformation](#), [FtpGetServerStatus](#), [FtpSetDirectoryFormat](#)

FtpGetStatus Function

```
INT WINAPI FtpGetStatus(  
    HCLIENT hClient  
);
```

The **FtpGetStatus** function returns the current status of the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is `FTP_ERROR`. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetStatus** function returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Constant	Description
1	<code>FTP_STATUS_IDLE</code>	The client is current idle and not sending or receiving data.
2	<code>FTP_STATUS_CONNECT</code>	The client is establishing a connection with the server.
3	<code>FTP_STATUS_READ</code>	The client is reading data from the server.
4	<code>FTP_STATUS_WRITE</code>	The client is writing data to the server.
5	<code>FTP_STATUS_DISCONNECT</code>	The client is disconnecting from the server.
6	<code>FTP_STATUS_OPENFILE</code>	The client is opening a data connection to the server.
7	<code>FTP_STATUS_CLOSEFILE</code>	The client is closing the data connection to the server.
8	<code>FTP_STATUS_GETFILE</code>	The client is downloading a file from the server.
9	<code>FTP_STATUS_PUTFILE</code>	The client is uploading a file to the server.
10	<code>FTP_STATUS_FILELIST</code>	The client is retrieving a file listing from the server.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session. To obtain status information about a file transfer, use the **FtpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

See Also

FtpGetTaskError Function

```
DWORD WINAPI FtpGetTaskError(  
    UINT nTaskId  
);
```

Return the last error code for the specified asynchronous task.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has completed successfully, this function returns a value of zero. A non-zero return value indicates an error has occurred.

Remarks

The **FtpGetTaskError** function returns the last error code associated with the specified asynchronous task. If the task completed successfully, the return value will be zero. If the task is still active, the function will return the error `ST_ERROR_TASK_ACTIVE`. If the task has been suspended, the function will return `ST_ERROR_TASK_SUSPENDED`. Any other value indicates that the task completed, but the operation has failed and the error code will specify the cause of the failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

See Also

[FtpTaskAbort](#), [FtpTaskResume](#), [FtpTaskSuspend](#), [FtpTaskWait](#)

FtpGetTaskId Function

```
UINT WINAPI FtpGetTaskId(  
    HCLIENT hClient  
);
```

Return the asynchronous task identifier associated with the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the client handle is not associated with an asynchronous task, the function will return a value of zero.

Remarks

The **FtpGetTaskId** function will return the task ID that is associated with a client session. This is a unique unsigned integer value that references the worker thread that was created to manage the asynchronous client session. This function should only be called within an event handler that is invoked by a background task that has been started using a function such as **FtpAsyncGetFile**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

See Also

[FtpTaskAbort](#), [FtpTaskDone](#), [FtpTaskResume](#), [FtpTaskSuspend](#), [FtpTaskWait](#)

FtpGetText Function

```
INT WINAPI FtpGetText(  
    HCLIENT hClient,  
    LPCTSTR lpszRemoteFile,  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);
```

The **FtpGetText** function copies the contents of a text file on the server to the specified string buffer.

Parameters

hClient

Handle to the client session.

lpszRemoteFile

A pointer to a string that specifies a text file on the server. The file pathing and naming conventions must be that of the host operating system.

lpszBuffer

A pointer to a string buffer which will contain the contents of the text file when the function returns. This buffer should be large enough to store the contents of the file, including a terminating null character. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. This value must be larger than zero. If this value is smaller than the actual size of the text file, the data returned will be truncated.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetText** function is used to download a text file and store the contents in a string buffer. This function will always set the file type to FILE_TYPE_ASCII before downloading the file, and will restore the default file type before the method returns. Because binary files can include embedded null characters, this function should only be used with known text files.

This function has been included as a convenience for applications that need to retrieve relatively small text files and manipulate the contents as a string. If the Unicode version of this function is called, the contents of the text file is automatically converted to a Unicode string. If the size of the file is unknown or the text file is very large, it is recommended that you use the **FtpGetData** or **FtpGetFile** functions.

If you use the **FtpGetFileSize** function to determine how large the string buffer should be prior to calling this function, it is important to be aware that the actual number of characters may differ based on the end-of-line conventions used by the host operating system. For example, if you call **FtpGetFileSize** to obtain the size of a text file on a UNIX system, the value will not be large enough to store the complete file because UNIX uses a single linefeed (LF) character to indicate the end-of-line, while a Windows system will use a carriage-return and linefeed (CRLF) pair. To accommodate this difference, you should always allocate extra memory for the string buffer to

store the additional end-of-line characters.

FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **FtpEnableEvents**, or by registering a callback function using the **FtpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **FtpGetTransferStatus** function.

Example

```
LPTSTR lpszBuffer = (LPTSTR)calloc(MAXFILESIZE, sizeof(TCHAR));

if (lpszBuffer == NULL)
    return;

nResult = FtpGetText(hClient, lpszRemoteFile, lpszBuffer, MAXFILESIZE);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpChangeDirectory](#), [FtpEnableEvents](#), [FtpGetData](#), [FtpGetFile](#), [FtpGetTransferStatus](#), [FtpPutData](#), [FtpPutFile](#), [FtpRegisterEvent](#), [FtpSetBufferSize](#)

FtpGetTimeout Function

```
INT WINAPI FtpGetTimeout(  
    HCLIENT hClient  
);
```

The **FtpGetTimeout** function returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is `FTP_ERROR`. To get extended error information, call **FtpGetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

See Also

[FtpConnect](#), [FtpIsReadable](#), [FtpIsWritable](#), [FtpProxyConnect](#), [FtpRead](#), [FtpSetTimeout](#), [FtpWrite](#)

FtpGetTransferStatus Function

```
INT WINAPI FtpGetTransferStatus(  
    HCLIENT hClient,  
    LPFTPTRANSFERSTATUS lpStatus  
);
```

The **FtpGetTransferStatus** function returns information about the current file transfer in progress.

Parameters

hClient

Handle to the client session.

lpStatus

A pointer to an **FTPTRANSFERSTATUS** structure which contains information about the status of the current file transfer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is **FTP_ERROR**. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetTransferStatus** function returns information about the current file transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no file currently being transferred, this function will return the status of the last successful transfer made by the client.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

If the option **FTP_OPTION_HIRES_TIMER** has been specified when connecting to the server, the values of the *dwTimeElapsed* and *dwTimeEstimated* members of the **FTPTRANSFERSTATUS** structure will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small files over a fast network connection.

If you are uploading or downloading large files which exceed 4GB, you should use the **FtpGetTransferStatusEx** function which returns the size as a 64-bit value.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

See Also

[FtpEnableEvents](#), [FtpGetFileStatus](#), [FtpGetTransferStatusEx](#), [FtpRegisterEvent](#), [FTPTRANSFERSTATUS](#), [FTPTRANSFERSTATUSEX](#)

FtpGetTransferStatusEx Function

```
INT WINAPI FtpGetTransferStatusEx(  
    HCLIENT hClient,  
    LPFTPTRANSFERSTATUS lpStatus  
);
```

The **FtpGetTransferStatusEx** function returns information about the current file transfer in progress. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpStatus

A pointer to an [FTPTRANSFERSTATUSEX](#) structure which contains information about the status of the current file transfer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `FTP_ERROR`. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpGetTransferStatusEx** function returns information about the current file transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no file currently being transferred, this function will return the status of the last successful transfer made by the client.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

If the option `FTP_OPTION_HIRES_TIMER` has been specified when connecting to the server, the values of the *dwTimeElapsed* and *dwTimeEstimated* members of the **FTPTRANSFERSTATUSEX** structure will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small files over a fast network connection.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

See Also

[FtpEnableEvents](#), [FtpGetFileStatus](#), [FtpGetTransferStatus](#), [FtpRegisterEvent](#), [FTPTRANSFERSTATUS](#), [FTPTRANSFERSTATUSEX](#)

FtpInitialize Function

```
BOOL WINAPI FtpInitialize(  
    LPCTSTR lpszLicenseKey,  
    LPINITDATA lpData  
);
```

The **FtpInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpData

Pointer to an INITDATA data structure. This parameter may be NULL if the initialization data for the library is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must insure that this function is called with a valid license key.

If the *lpData* argument is specified, it must point to an INITDATA structure which has been initialized by setting the *dwSize* member to the size of the structure. All other structure members should be set to zero. If the function is successful, then the INITDATA structure will be filled with identifying information about the library.

Although it is only required that **FtpInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **FtpUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **FtpInitialize** or **FtpUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

FtpIsBlocking Function

```
BOOL WINAPI FtpIsBlocking(  
    HCLIENT hClient  
);
```

The **FtpIsBlocking** function is used to determine if the client is currently performing a blocking operation.

Parameters

hClient

Handle to the client session.

Return Value

If the client is performing a blocking operation, the function returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the function returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking function may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **FtpIsBlocking** function can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpCancel](#), [FtpGetStatus](#), [FtpIsConnected](#), [FtpIsReadable](#), [FtpIsWritable](#), [FtpRead](#), [FtpWrite](#)

FtpIsConnected Function

```
BOOL WINAPI FtpIsConnected(  
    HCLIENT hClient  
);
```

The **FtpIsConnected** function is used to determine if the client is currently connected to a server.

Parameters

hClient

Handle to the client session.

Return Value

If the client is connected to a server, the function returns a non-zero value. If the client is not connected, or the client handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

See Also

[FtpGetStatus](#), [FtpIsBlocking](#), [FtpIsReadable](#), [FtpIsWritable](#)

FtpIsReadable Function

```
BOOL WINAPI FtpIsReadable(  
    HCLIENT hClient,  
    INT nTimeout,  
    LPDWORD lpdwAvail  
);
```

The **FtpIsReadable** function is used to determine if data is available to be read from the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the function returns a non-zero value. If the client cannot read any data, the function returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

See Also

[FtpGetStatus](#), [FtpIsBlocking](#), [FtpIsConnected](#), [FtpIsWritable](#), [FtpRead](#)

FtpIsWritable Function

```
BOOL WINAPI FtpIsWritable(  
    HCLIENT hClient,  
    INT nTimeout  
);
```

The **FtpIsWritable** function is used to determine if data can be written to the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the function returns a non-zero value. If the client cannot write any data, the function returns zero.

Remarks

Although this function can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

See Also

[FtpGetStatus](#), [FtpIsBlocking](#), [FtpIsConnected](#), [FtpIsReadable](#), [FtpWrite](#)

FtpLogin Function

```
INT WINAPI FtpLogin(  
    HCLIENT hClient,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszAccount  
);
```

The **FtpLogin** function authenticates the specified user in on the server. This function must be called after the connection has been established, and before attempting to transfer files or perform any other function on the server.

Parameters

hClient

Handle to the client session.

lpszUserName

Points to a string that specifies the user name to be used to authenticate the current client session. If this parameter is NULL or an empty string, then the login is considered to be anonymous.

lpszPassword

Points to a string that specifies the password to be used to authenticate the current client session. This parameter may be NULL or an empty string if no password is required for the specified user, or if no username has been specified.

lpszAccount

Points to a string that specifies the account name to be used to authenticate the current client session. This parameter may be NULL or an empty string if no account name is required for the specified user.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

Some public FTP servers support anonymous logins, where a username and password are not required to access the server. In this case, both the *lpszUserName* and *lpszPassword* parameters can be NULL or specify empty strings. In most cases, access to the server using an anonymous login is restricted, with clients only having permission to download files. Servers may also restrict the maximum number of anonymous sessions that may be logged in at one time.

This function should only be used after calling the **FtpLogout** function, enabling you to log in as another user during the same session. Not all servers will permit a client to change user credentials during the same session. In most cases, it is preferable to disconnect from the server and re-connect using the new credentials rather than using this function.

This function is not supported with secure connections using the SSH protocol.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpConnect](#), [FtpLogout](#), [FtpProxyConnect](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

FtpLogout Function

```
INT WINAPI FtpLogout(  
    HCLIENT hClient  
);
```

The **FtpLogout** function logs out the user associated with the current client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpLogout** function is used when the client wants to re-authenticate using a new username and password. Before any further action may be taken, other than disconnecting from the server, the **FtpLogin** function must be called to re-authenticate the client.

It is not necessary to call this function prior to disconnecting from the server because the user current user is automatically logged out when the **FtpDisconnect** function is called.

This function is not supported with secure connections using the SSH protocol.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpConnect](#), [FtpDisconnect](#), [FtpLogin](#)

FtpMountStructure Function

```
INT WINAPI FtpMountStructure(  
    HCLIENT hClient,  
    LPCTSTR lpszFileSystem  
);
```

The **FtpMountStructure** function mounts a different file system or other directory data structure on the server.

Parameters

hClient

Handle to the client session.

lpszFileSystem

A pointer to a string which specifies the file system to mount on the server.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function sends the SMNT command to the server, which may not be supported on some platforms. Use of this command typically requires that the user have administrator privileges on the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpCreateDirectory](#), [FtpRemoveDirectory](#)

FtpOpenDirectory Function

```
INT WINAPI FtpOpenDirectory(  
    HCLIENT hClient,  
    LPCTSTR lpszDirectory  
);
```

The **FtpOpenDirectory** function opens the specified directory on the server.

Parameters

hClient

Handle to the client session.

lpszDirectory

Pointer to the name of the directory that will be opened. The format of the directory name must match the filename conventions used by the server. If a NULL pointer or an empty string is specified, then the current working directory is opened.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpOpenDirectory** function opens the specified directory on the server using the LIST command. The contents of the directory can be read using the **FtpGetFirstFile** and **FtpGetNextFile** functions. The directory listing is returned on the data channel in one of several different formats. The library can recognize listing formats generated by UNIX, VMS and Windows servers, as well as those of other servers which emulate one of those common formats. Once the complete directory listing has been read, the directory must be closed by calling the **FtpCloseDirectory** function.

Because the directory listing is returned on the data channel, a file transfer cannot be performed while the directory is in the process of being read by the client. Applications which need to collect a list of files to download should first open the directory, read the contents and store the file names in an array. After the directory has been closed, the application can then start transferring the files to the local system.

Some servers may not support file listings for any directory other than the current working directory. If an error is returned when specifying a directory name, try changing the current working directory using the **FtpChangeDirectory** function and then call this function again, passing NULL or an empty string as the *lpszDirectory* parameter.

To obtain a list of all files in a directory using a single function call, use the **FtpEnumFiles** function. If the server lists files in a format that is not recognized by the library, the **FtpGetFileList** function can be used to obtain an unparsed file listing from the server.

Example

```
if (FtpOpenDirectory(hClient, NULL) != FTP_ERROR)  
{  
    FTPFILESTATUS ftpFile;  
    BOOL bResult;  
  
    bResult = FtpGetFirstFile(hClient, &ftpFile);  
    while (bResult)
```

```
{  
    // The ftpFile structure contains information about the file  
    bResult = FtpGetNextFile(hClient, &ftpFile);  
}  
  
FtpCloseDirectory(hClient);  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpChangeDirectory](#), [FtpCloseDirectory](#), [FtpEnumFiles](#), [FtpGetDirectoryFormat](#), [FtpGetFileList](#),
[FtpGetFileStatus](#), [FtpGetFirstFile](#), [FtpGetNextFile](#), [FtpSetDirectoryFormat](#)

FtpOpenFile Function

```
INT WINAPI FtpOpenFile(  
    HCLIENT hClient,  
    LPCTSTR lpszFileName,  
    DWORD dwOpenMode,  
    DWORD dwOffset  
);
```

The **FtpOpenFile** function creates or opens the specified file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file to create or open. The file pathing and name conventions must be that of the server.

dwOpenMode

Specifies the type of access to the file. An application can open a file for reading, create a new file or append data to an existing file. This parameter should be one of the following values.

Constant	Description
FTP_FILE_READ	The file is opened for reading on the server. A data channel is created and the contents of the file are returned to the client.
FTP_FILE_WRITE	The file is opened for writing on the server. If the file does not exist, it will be created. If it does exist, it will be overwritten.
FTP_FILE_APPEND	The file is opened for writing on the server. All data will be appended to the end of the file.

dwOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset using FTP requires that the server support the REST command to restart transfers.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

Only one file may be opened at a time for each client session. Attempting to perform an action such as uploading or downloading another file while a file is currently open will result in an error. Typically this indicates that the application failed to call the **FtpCloseFile** function.

It is strongly recommended that most applications use the **FtpGetFile** or **FtpPutFile** functions to perform file transfers. These functions are easier to use, and have internal optimizations that improves the overall data transfer rate when compared to implementing the file transfer code in your own application.

When a file is created on the server, the file ownership and access rights are determined by the server. Some servers may provide a method to change these attributes through site-specific commands. Refer to the server's operating system documentation for more information about what commands may be available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpCloseFile](#), [FtpGetFile](#), [FtpGetFileSize](#), [FtpGetFileTime](#), [FtpPutFile](#), [FtpSetFileTime](#)

FtpOpenFileEx Function

```
INT WINAPI FtpOpenFileEx(  
    HCLIENT hClient,  
    LPCTSTR lpszFileName,  
    DWORD dwOpenMode,  
    DWORD dwReserved,  
    ULARGE_INTEGER uiOffset  
);
```

The **FtpOpenFileEx** function creates or opens the specified file on the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file to create or open. The file pathing and name conventions must be that of the server.

dwOpenMode

Specifies the type of access to the file. An application can open a file for reading, create a new file or append data to an existing file. This parameter should be one of the following values.

Constant	Description
FTP_FILE_READ	The file is opened for reading on the server. A data channel is created and the contents of the file are returned to the client.
FTP_FILE_WRITE	The file is opened for writing on the server. If the file does not exist, it will be created. If it does exist, it will be overwritten.
FTP_FILE_APPEND	The file is opened for writing on the server. All data will be appended to the end of the file.

dwReserved

An unsigned integer value that is reserved for future use. This parameter should always have a value of zero.

uiOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset using FTP requires that the server support the REST command to restart transfers.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

Only one file may be opened at a time for each client session. Attempting to perform an action such as uploading or downloading another file while a file is currently open will result in an error.

Typically this indicates that the application failed to call the **FtpCloseFile** function.

It is strongly recommended that most applications use the **FtpGetFileEx** or **FtpPutFileEx** functions to perform file transfers. These functions are easier to use, and have internal optimizations that improves the overall data transfer rate when compared to implementing the file transfer code in your own application.

When a file is created on the server, the file ownership and access rights are determined by the server. Some servers may provide a method to change these attributes through site-specific commands. Refer to the server's operating system documentation for more information about what commands may be available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpCloseFile](#), [FtpGetFileEx](#), [FtpGetFileSizeEx](#), [FtpGetFileTime](#), [FtpPutFileEx](#), [FtpSetFileTime](#)

FtpProxyConnect Function

```
HCLIENT WINAPI FtpProxyConnect(  
    UINT nProxyType,  
    LPCTSTR LpszProxyHost,  
    UINT nProxyPort,  
    LPCTSTR LpszProxyUser,  
    LPCTSTR LpszProxyPassword,  
    LPCTSTR LpszRemoteHost,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSECURITYCREDENTIALS LpCredentials  
);
```

The **FtpProxyConnect** function establishes a connection through a proxy server and defines security related options to be used. Four basic proxy server types are recognized, and the library will automatically negotiate with the server to establish a connection through the proxy server to the destination server.

Parameters

nProxyType

An identifier which specifies the type of proxy server that is being connected to. This value must be defined as one of the following values:

Constant	Description
FTP_PROXY_NONE	This value specifies that no proxy server is being used. In this case, the FtpConnect function is called directly, ignoring the proxy parameters.
FTP_PROXY_USER	This value specifies that the client is not logged into the proxy server. The USER command is sent in the format username@ftpsite followed by the password. This is the format used with the Gauntlet proxy server.
FTP_PROXY_LOGIN	This value specifies that the client is logged into the proxy server. The USER command is then sent in the format username@ftpsite followed by the password. This is the format used by the InterLock proxy server.
FTP_PROXY_OPEN	This value specifies that the client is not logged into the proxy server. The OPEN command is sent specifying the host name, followed by the username and password.
FTP_PROXY_SITE	This value specifies that the client is logged into the server. The SITE command is sent, specifying the host name, followed by the username and the password.
FTP_PROXY_OTHER	This special proxy type specifies that another, undefined proxy server is being used. The client connects to the proxy host, but does not attempt to authenticate the client. The application is responsible for negotiating with the proxy server, typically using the FtpCommand function to send specific command sequences.

lpszProxyHost

A pointer to the name of the proxy server to connect through; this may be a fully-qualified domain name or an IP address.

lpszProxyPort

The port number the proxy server is listening on; a value of zero specifies that the default port number should be used.

lpszProxyUser

A pointer to the user name used to authenticate the client on the proxy server. Not all proxy servers require this information; it is recommended that you consult the proxy server documentation to determine if a username is required.

lpszProxyPassword

A pointer to the password used to authenticate the client on the proxy server. Not all proxy servers require this information; it is recommended that you consult the proxy server documentation to determine if a password is required.

lpszRemoteHost

A pointer to the name of the server that you want to connect to, through the proxy server.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
FTP_OPTION_PASSIVE	This option specifies the client should attempt to establish the data connection with the server. When the client uploads or downloads a file, normally the server establishes a second connection back to the client which is used to transfer the file data. However, if the local system is behind a firewall or a NAT router, the server may not be able to create the data connection and the transfer will fail. By specifying this option, it forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_NOAUTH	This option specifies the server does not require authentication, or that it requires an alternate

	<p>authentication method. When this option is used, the client connection is flagged as authenticated as soon as the connection to the server has been established. Note that using this option to bypass authentication may result in subsequent errors when attempting to retrieve a directory listing or transfer a file. It is recommended that you consult the technical reference documentation for the server to determine its specific authentication requirements.</p>
FTP_OPTION_KEEPALIVE	<p>This option specifies the client should attempt to keep the connection with the server active for an extended period of time. It is important to note that regardless of this option, the server may still choose to disconnect client sessions that are holding the command channel open but are not performing file transfers.</p>
FTP_OPTION_VIRTUALHOST	<p>This option specifies the server supports virtual hosting, where multiple domains are hosted by a server using the same external IP address. If this option is enabled, the client will send the HOST command to the server upon establishing a connection.</p>
FTP_OPTION_VERIFY	<p>This option specifies that file transfers should be automatically verified after the transfer has completed. If the server supports the XMD5 command, the transfer will be verified by calculating an MD5 hash of the file contents. If the server does not support the XMD5 command, but does support the XCRC command, the transfer will be verified by calculating a CRC32 checksum of the file contents. If neither the XMD5 or XCRC commands are supported, the transfer is verified by comparing the size of the file. Automatic file verification is only performed for binary mode transfers because of the end-of-line conversion that may occur when text files are uploaded or downloaded.</p>
FTP_OPTION_TRUSTEDSITE	<p>This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.</p>
FTP_OPTION_SECURE	<p>This option specifies the client should attempt to establish a secure connection with the server. This option is the same as specifying FTP_OPTION_SECURE_IMPLICIT which immediately performs the SSL/TLS protocol negotiation when the connection is established.</p>

FTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to immediately establish secure SSL/TLS connection with the server. This option is typically used when connecting to a server on port 990, which is the default port number used for FTPS.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should establish a standard connection to the server and then use the AUTH command to negotiate an explicit secure connection. This option is typically used when connecting to the server on ports other than 990.
FTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
FTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established. This option also forces all connections to be outbound and enables the firewall compatibility features in the client.
FTP_OPTION_KEEPALIVE_DATA	This option specifies the client should attempt to keep the control connection active during a file transfer. Normally, when a data transfer is in progress, no additional commands are issued on the control channel until the transfer completes. Specifying this option automatically enables the FTP_OPTION_KEEPALIVE option and forces the client to continue to issue NOOP commands during the file transfer. This option only applies to FTP and FTPS connections and has no effect on connections using SFTP (SSH).
FTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
FTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The

	application is responsible for ensuring that access to the handle is synchronized across multiple threads.
FTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller files being uploaded or downloaded using fast network connections.
FTP_OPTION_TLS_REUSE	This option specifies that TLS session reuse should be enabled for secure connections. This option is only supported on Windows 8.1 or Windows Server 2012 R2 and later platforms, and it should only be used when explicitly required by the server. This option is not compatible with servers built using OpenSSL 1.0.2 and earlier versions which do not provide Extended Master Secret (EMS) support as outlined in RFC7627.

lpCredentials

A pointer to a [SECURITYCREDENTIALS](#) structure. This parameter should be NULL if the connection is not secure or when client credentials are not required. Most servers do not require a client certificate to establish a secure connection. However, if the server does require a client certificate, the structure members *dwSize*, *lpzCertStore* and *lpzCertName* must be defined. Undefined structure members must be initialized to a value of zero or NULL and the *dwSize* member must be initialized to the size of the **SECURITYCREDENTIALS** structure.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **FtpGetLastError**.

Remarks

To prevent this function from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **FtpProxyConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

The username and password that is used to authenticate the client with the proxy server are not the same as those used to login to the target server. Once a connection has been established with the proxy server, the client must call the **FtpLogin** function to actually login to the server and begin a file transfer.

If the FTP_OPTION_KEEPALIVE option is specified, a background worker thread will be created to monitor the command channel and periodically send NOOP commands to the server if no commands have been sent recently. This can prevent the server from terminating the client connection during idle periods where no commands are being issued. However, it is important to keep in mind that many servers can be configured to also limit the total amount of time a client can be connected to the server, as well as the amount of time permitted between file transfers. If the server does not respond to the NOOP command, this option will be automatically disabled for the remainder of the client session.

If the FTP_OPTION_SECURE_EXPLICIT option is specified, the client will establish a standard connection to the server and send the AUTH TLS command to the server. If the server does not

accept this command, it will then send the AUTH SSL command. If both commands are rejected by the server, an explicit SSL session cannot be established. By default, both the command and data channels will be encrypted when a secure connection is established. To change this, use the **FtpSetChannelMode** function.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **FtpAttachThread** function.

Specifying the FTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpConnect](#), [FtpCreateSecurityCredentials](#), [FtpDeleteSecurityCredentials](#), [FtpDisconnect](#), [FtpGetSecurityInformation](#), [FtpInitialize](#), [FtpLogin](#), [FtpSetChannelMode](#)

FtpPutData Function

```
INT WINAPI FtpPutData(  
    HCLIENT hClient,  
    LPCTSTR lpszFileName,  
    LPVOID lpvBuffer,  
    DWORD dwLength,  
    DWORD dwReserved  
);
```

The **FtpPutData** function transfers the contents of the specified buffer to a file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

A pointer to a string that specifies the file on the server that will be created, overwritten or appended to. The file naming conventions must be that of the host operating system.

lpvBuffer

A pointer to the data that will be copied to the server and stored in the specified file.

dwLength

The number of bytes to copy from the buffer.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

If the *lpvBuffer* parameter is pointing to a Unicode string, it is important to note that the value of the *dwLength* parameter should specify the number of bytes, not the number of characters. When using UTF-16, each character is two bytes long and therefore the length of the buffer is effectively double the length of the string. Because Unicode strings can contain null characters, you must also set the current file type to FILE_TYPE_IMAGE prior to calling this function.

The **FtpPutText** function can be used to create a text file from the contents of a string.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **FtpEnableEvents**, or by registering a callback function using the **FtpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **FtpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpChangeDirectory](#), [FtpEnableEvents](#), [FtpGetData](#), [FtpGetFile](#), [FtpGetTransferStatus](#), [FtpPutFile](#),
[FtpRegisterEvent](#), [FtpSetBufferSize](#)

FtpPutFile Function

```
INT WINAPI FtpPutFile(  
    HCLIENT hClient,  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszRemoteFile,  
    DWORD dwOptions,  
    DWORD dwOffset  
);
```

The **FtpPutFile** function transfers the specified file on the local system to the server.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A pointer to a string that specifies the file that will be transferred from the local system. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A pointer to a string that specifies the file on the server that will be created, overwritten or appended to. The file naming conventions must be that of the host operating system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Constant	Description
FTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the remote file exists, it will be overwritten with the contents of the uploaded file.
FTP_TRANSFER_APPEND	This option specifies that if the remote file exists, the contents of the local file is appended to the remote file. If the remote file does not exist, it is created.

dwOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **FtpEnableEvents**, or by registering a callback function using the **FtpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **FtpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpChangeDirectory](#), [FtpEnableEvents](#), [FtpGetData](#), [FtpGetFile](#), [FtpGetMultipleFiles](#), [FtpGetTransferStatus](#), [FtpPutData](#), [FtpPutMultipleFiles](#), [FtpRegisterEvent](#), [FtpSetBufferSize](#), [FtpVerifyFile](#)

FtpPutFileEx Function

```
INT WINAPI FtpPutFileEx(  
    HCLIENT hClient,  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszRemoteFile,  
    DWORD dwOptions,  
    ULARGE_INTEGER uiOffset  
);
```

The **FtpPutFileEx** function transfers the specified file on the local system to the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A pointer to a string that specifies the file that will be transferred from the local system. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A pointer to a string that specifies the file on the server that will be created, overwritten or appended to. The file naming conventions must be that of the host operating system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Constant	Description
FTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the remote file exists, it will be overwritten with the contents of the uploaded file.
FTP_TRANSFER_APPEND	This option specifies that if the remote file exists, the contents of the local file is appended to the remote file. If the remote file does not exist, it is created.

uiOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the FTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **FtpEnableEvents**, or by registering a callback function using

the **FtpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **FtpGetTransferStatusEx** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpChangeDirectory](#), [FtpEnableEvents](#), [FtpGetData](#), [FtpGetFileEx](#), [FtpGetMultipleFiles](#), [FtpGetTransferStatusEx](#), [FtpPutData](#), [FtpPutMultipleFiles](#), [FtpRegisterEvent](#), [FtpSetBufferSize](#), [FtpVerifyFile](#)

FtpPutMultipleFiles Function

```
INT WINAPI FtpPutMultipleFiles(  
    HCLIENT hClient,  
    LPCTSTR lpszLocalDirectory,  
    LPCTSTR lpszRemoteDirectory,  
    LPCTSTR lpszFileMask,  
    DWORD dwReserved  
);
```

The **FtpPutMultipleFiles** function copies one or more files from the local host to the server, using the specified wildcard.

Parameters

hClient

Handle to the client session.

lpszLocalDirectory

Pointer to a string which specifies the local directory where the files will be copied from. A NULL pointer or empty string specifies that files should be copied from the current working directory.

lpszRemoteDirectory

Pointer to a string which specifies the remote directory where the files will be copied to. A NULL pointer or empty string specifies that the files should be copied to the current working directory on the server.

lpszFileMask

Pointer to a string which specifies the files that are to be copied from the local system to the server. The file mask should follow the Windows conventions used for wildcard file matches.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpPutMultipleFiles** function is used to transfer files from the local host to the server which match a specified wildcard file mask. All files will be transferred using the current file type as specified by the **FtpSetFileType** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpChangeDirectory](#), [FtpGetFile](#), [FtpGetMultipleFiles](#), [FtpPutFile](#)

FtpPutText Function

```
INT WINAPI FtpPutText(  
    HCLIENT hClient,  
    LPCTSTR lpszRemoteFile,  
    LPCTSTR lpszBuffer  
);
```

The **FtpPutText** function creates a text file on the server using the contents of a string buffer.

Parameters

hClient

Handle to the client session.

lpszRemoteFile

A pointer to a string that specifies the text file on the server that will be created or overwritten. The file pathing and name conventions must be that of the server.

lpszBuffer

A pointer to a string that contains the text that will be stored in the file.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is `FTP_ERROR`. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpPutText** function is used to create a text file on the server from the contents of a string. If the specified file already exists on the server, its contents will be overwritten. This function will always set the file type to `FILE_TYPE_ASCII` before creating the file, and will restore the default file type before the method returns.

If the Unicode version of this function is called, the string will be converted to ASCII and then uploaded to the server. If you wish to store the contents of the string as Unicode on the server, you must set the current file type to `FILE_TYPE_IMAGE` and use the **FtpPutData** function. This function should never be used to upload binary data.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the `FTP_EVENT_PROGRESS` event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **FtpEnableEvents**, or by registering a callback function using the **FtpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **FtpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpChangeDirectory](#), [FtpEnableEvents](#), [FtpGetText](#), [FtpGetTransferStatus](#), [FtpPutData](#), [FtpPutFile](#),

FtpRead Function

```
INT WINAPI FtpRead(  
    HCLIENT hClient,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **FtpRead** function reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hClient

Handle to the client session.

lpBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

When **FtpRead** is called and the client is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpEnableEvents](#), [FtpGetData](#), [FtpGetFile](#), [FtpIsBlocking](#), [FtpIsReadable](#), [FtpIsWritable](#), [FtpRegisterEvent](#), [FtpWrite](#)

FtpRegisterEvent Function

```
INT WINAPI FtpRegisterEvent(  
    HCLIENT hClient,  
    UINT nEventId,  
    FTPEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

The **FtpRegisterEvent** function registers an event handler for the specified event.

Parameters

hClient

Handle to the client session.

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Constant	Description
FTP_EVENT_CONNECT	The control connection to the server has completed.
FTP_EVENT_DISCONNECT	The server has closed the control connection to the client. The client should read any remaining data and disconnect.
FTP_EVENT_OPENFILE	The data connection to the server has completed.
FTP_EVENT_CLOSEFILE	The server has closed the data connection to the client. The client should read any remaining data and close the data channel.
FTP_EVENT_READFILE	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_WRITEFILE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
FTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
FTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
FTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.

FTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.
FTP_EVENT_GETFILE	This event is generated when a file download has completed. If multiple files are being downloaded, this event will be generated for each file.
FTP_EVENT_PUTFILE	This event is generated when a file upload has completed. If multiple files are being uploaded, this event will be generated for each file.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpRegisterEvent** function associates a callback function with a specific event. The event handler is an **FtpEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This function is typically used to register an event handler that is invoked while a file is being uploaded or downloaded. The FTP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one FTP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpDisableEvents](#), [FtpEnableEvents](#), [FtpEventProc](#), [FtpFreezeEvents](#)

FtpRegisterFileType Function

```
BOOL WINAPI FtpRegisterFileType(  
    HCLIENT hClient,  
    LPCTSTR lpszExtension,  
    UINT nFileType  
);
```

The **FtpRegisterFileType** function associates a file name extension with a specific file type.

Parameters

hClient

Handle to the client session. If this parameter is `INVALID_CLIENT`, the file type is registered for all client sessions in the current process. If the parameter specifies a valid client session, then the association is made only for that specific session.

lpszExtension

A pointer to a null terminated string which specifies the file name extension. If this parameter is `NULL` or points to an empty string, the default file type will be changed for the client session.

nFileType

Specifies the type of file associated with the file extension. This parameter can be one of the following values.

Value	Description
<code>FILE_TYPE_ASCII</code>	The file is a text file using the ASCII character set. For those servers which mark the end of a line with characters other than a carriage return and linefeed, it will be converted to the native client format. This is the file type used for directory listings.
<code>FILE_TYPE_EBCDIC</code>	The file is a text file using the EBCDIC character set. Local files will be converted to EBCDIC when sent to the server. Remote files will be converted to the native ASCII character set when retrieved from the server.
<code>FILE_TYPE_IMAGE</code>	The file is a binary image and no data conversion of any type is performed on the file. This is typically the default file type for data file transfers. If the type of file that is being transferred is unknown, this file type should always be used.

Return Value

If the function succeeds, the return value non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpRegisterFileType** function is used to associate specific file types with file name extensions. The library has an internal list of standard text file extensions which it automatically recognizes. This method can be used to extend or modify that list for the client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpOpenFile](#), [FtpRegisterFileType](#), [FtpSetFileMode](#), [FtpSetFileStructure](#), [FtpSetFileType](#),
[FtpSetPassiveMode](#)

FtpRemoveDirectory Function

```
INT WINAPI FtpRemoveDirectory(  
    HCLIENT hClient,  
    LPCTSTR lpszDirectory  
);
```

The **FtpRemoveDirectory** function removes the specified directory on the server.

Parameters

hClient

Handle to the client session.

lpszDirectory

Points to a string that specifies the name of the directory. The file pathing and name conventions must be that of the server.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function uses the RMD command to create the directory. The user must have the appropriate permission to remove the specified directory. Most servers will not permit you to remove a directory if it contains one or more files.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpChangeDirectory](#), [FtpCreateDirectory](#), [FtpDeleteFile](#), [FtpGetDirectory](#)

FtpRenameFile Function

```
INT WINAPI FtpRenameFile(  
    HCLIENT hClient,  
    LPCTSTR lpszOldFileName,  
    LPCTSTR lpszNewFileName  
);
```

The **FtpRenameFile** function renames the specified file on the server. The file must exist, and the current user must have the appropriate permission to change the file name.

Parameters

hClient

Handle to the client session.

lpszOldFileName

Points to a string that specifies the name of the remote file to rename. The file pathing and name conventions must be that of the server.

lpszNewFileName

Points to a string that specifies the new name for the remote file. The file pathing and name conventions must be that of the server.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function causes two separate commands to be sent to the server, RNFR and RNT0. If either command fails, the function will fail and return an error code.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpDeleteFile](#), [FtpGetFile](#), [FtpPutFile](#)

FtpReset Function

```
INT WINAPI FtpReset(  
    HCLIENT hClient  
);
```

The **FtpReset** function resets the client state and resynchronizes with the server. This function is typically called after an unexpected error has occurred, or an operation has been canceled.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The client cannot be reset while a file transfer is in progress or if the client is in a blocked state. To abort a file transfer, use the **FtpCancel** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpCancel](#)

FtpSetActivePorts Function

```
INT WINAPI FtpSetActivePorts(  
    HCLIENT hClient,  
    UINT nLowPort,  
    UINT nHighPort  
);
```

The **FtpSetActivePorts** function changes the range of local port numbers used for active mode file transfers.

Parameters

hClient

Handle to the client session.

nLowPort

An unsigned integer that specifies the low port number.

lpnHighPort

An unsigned integer that specifies the high port number.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

This function is used to modify the range of local port numbers used for active mode file transfers. When using active mode, the client listens for an inbound connection from the server rather than establishing an outbound connection for the data transfer. In most cases, passive mode transfers are preferred because they mitigate potential compatibility issues with firewalls and NAT routers.

If active mode transfers are required, the default port range used when listening for the server connection is between 1024 and 5000. This is the standard range of ephemeral ports used by the Windows operating system. However, under some circumstances that range of ports may be too small, or a firewall may be configured to deny inbound connections on ephemeral ports. In that case, the **FtpSetActivePorts** function can be used to specify a different range of port numbers.

While it is technically permissible to assign the low and high port numbers to the same value, effectively specifying a single active port number, this is not recommended as it can cause the transfer to fail unexpectedly if multiple file transfers are performed. A minimum range of at least 1000 ports is recommended. For example, if you specify a low port value of 40000 then it is recommended that the high port value be at least 41000. The maximum port value is 65535.

To determine the current range of active port numbers being used, call the **FtpGetActivePorts** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

FtpSetBufferSize Function

```
INT WINAPI FtpSetBufferSize(  
    HCLIENT hClient,  
    INT nBufferSize  
);
```

The **FtpSetBufferSize** function sets the size in bytes of an internal buffer that will be used during data transfers.

Parameters

hClient

Handle to the client session.

nBufferSize

The size of an internal buffer, in bytes. Any value greater than or equal to zero is acceptable. If *nBufferSize* is zero, then the default value of 4096 will be used. If *nBufferSize* is less than 256 bytes, the buffer size will be set to 256. The maximum value of *nBufferSize* is 1048576 (1Mb).

Return Value

If the function succeeds, the return value is the size of the internal buffer that will be used. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The speed of data transfers, particularly on uploads, may be sensitive to network type and configuration, and the size of the internal buffer used for data transfers. The default size of this buffer will result in good performance for a wide range of network characteristics. A larger buffer will not necessarily result in better performance. For example, a multiple of 1460, which is the typical Maximum Transmission Unit (MTU), may be optimal in many situations.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetBufferSize](#), [FtpGetData](#), [FtpGetFile](#), [FtpPutData](#), [FtpPutFile](#)

FtpSetChannelMode Function

```
INT WINAPI FtpSetChannelMode(  
    HCLIENT hClient,  
    INT nChannel,  
    INT nMode  
);
```

The **FtpSetChannelMode** function changes the security mode for the specified communication channel.

Parameters

hClient

Handle to the client session.

nChannel

An integer value which specifies which channel to return information for. It may be one of the following values:

Constant	Description
FTP_CHANNEL_COMMAND	Change information for the command channel. This is the communication channel used to send commands to the server and receive command result and status information from the server.
FTP_CHANNEL_DATA	Change information for the data channel. This is the communication channel used to send or receive data during a file transfer.

nMode

An integer value which specifies the new mode for the specified channel. It may be one of the following values:

Constant	Description
FTP_CHANNEL_CLEAR	Data sent and received on this channel should not be encrypted.
FTP_CHANNEL_SECURE	Data sent and received on this channel should be encrypted. Specifying this option requires that a secure connection has already been established with the server.

Return Value

If the function succeeds, the return value is the previous mode for the specified channel. If the function fails, it will return FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpSetChannelMode** function is used to change the default mode for the specified channel, and is typically used to control whether or not data is encrypted during a file transfer. If a standard, non-secure connection has been established with the server, an error will be returned if you specify the FTP_CHANNEL_SECURE mode for either channel.

If you have established a secure connection and then specify the FTP_CHANNEL_CLEAR mode for

the command channel, the client will send the CCC command to the server to indicate that commands should no longer be encrypted. If the server does not support this command, an error will be returned and the channel mode will remain unchanged. Once the command channel has been changed to clear mode, it cannot be changed back to secure mode. You must disconnect and re-connect to the server if you want to resume sending commands over an encrypted channel.

Changing the mode for the data channel requires that the server support the PROT command. If this command is not supported by the server, the function will fail and the channel mode will remain unchanged.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[FtpGetChannelMode](#)

FtpSetDirectoryFormat Function

```
INT WINAPI FtpSetDirectoryFormat(  
    HCLIENT hClient,  
    INT nFormatId  
);
```

The **FtpSetDirectoryFormat** function is used to specify the format used by the server when returning a list of files. The format type is used internally by the library when parsing the file list returned by the server.

Parameters

hClient

Handle to the client session.

nFormatId

An identifier used to specify the format of the file list returned by the server. The following values are recognized:

Constant	Description
FTP_DIRECTORY_AUTO	This value specifies that the library should automatically determine the format of the file lists returned by the server. It is recommended that most applications use this value and allow the library to automatically determine the appropriate file listing format used by the server.
FTP_DIRECTORY_UNIX	This value specifies that the server returns file lists in the format commonly used by UNIX servers. Note that many servers can be configured to return file listings in this format, even if they are not actually a UNIX based platform. Consult the technical reference documentation for your server for more information.
FTP_DIRECTORY_MSDOS	This value specifies that the server returns file lists in the format commonly used by MS-DOS based systems. This includes Windows IIS servers. Long file names will be returned if supported by the underlying filesystem, such as NTFS or FAT32.
FTP_DIRECTORY_VMS	This value specifies that the server returns file lists in the format commonly used by VMS servers. Note that VMS servers can be configured to return a standard UNIX style listing in addition to the default VMS format.
FTP_DIRECTORY_STERLING_1	This value specifies that the server returns file listings in a proprietary format used by the Sterling server, which is used for EDI (Electronic Data Interchange) applications. This format uses a 13 byte status code.
FTP_DIRECTORY_STERLING_2	This value specifies that the server returns file listings in a proprietary format used by the Sterling server, which is used for EDI (Electronic Data Interchange)

	applications. This format uses a 10 byte status code.
FTP_DIRECTORY_NETWARE	This value specifies that the server returns file listings in a proprietary format used by NetWare servers. The format is similar to UNIX style listings except that file access and permissions are indicated by letter codes enclosed in brackets. This is the default format selected if the server identifies itself as a NetWare system.
FTP_DIRECTORY_MLSD	This value specifies that the server should return file listings in a machine-independent format as defined by RFC 3659. This format specifies file information as a sequence of name/value pairs, with the same format being used regardless of the operating system that the server is hosted on. Note that not all servers support this format, and some proxy servers may reject the command even if the remote server supports its use.

Return Value

If the function succeeds, the return value identifies the file list format used by the server. If the function fails, the return value is `FTP_ERROR`. To get extended error information, call **FtpGetLastError**.

Remarks

This function should only be used when the library cannot automatically determine the directory format returned by the server. To determine the format used by a server after a file list has been retrieved, use the **FtpGetDirectoryFormat** function.

The default directory format is determined both by the server's operating system and by analyzing the format of the data returned by the server. If the library is unable to automatically determine the format, it will attempt to parse the list of files as though it is a UNIX style listing.

If the `FTP_DIRECTORY_MLSD` format is specified, the file information returned by the server may differ from the default output of the `LIST` command. For example, on a UNIX based FTP server, the output of the `LIST` command is typically the same format that is used by the `/bin/lis` command, where file names are sorted and hidden files are not listed. However, the `MLSD` command may return an unsorted list of files that includes hidden files and directories.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpCloseDirectory](#), [FtpGetDirectoryFormat](#), [FtpGetFileStatus](#), [FtpGetFirstFile](#), [FtpGetNextFile](#), [FtpOpenDirectory](#)

FtpSetFeatures Function

```
DWORD WINAPI FtpSetFeatures(  
    HCLIENT hClient,  
    DWORD dwFeatures  
);
```

The **FtpSetFeatures** function specifies the server features available to the client.

Parameters

hClient

Handle to the client session.

dwFeatures

An unsigned integer that specifies one or more features. Refer to the documentation for the **FtpGetFeatures** function for a list of available features.

Return Value

If the function succeeds, the return value specifies the features that were previously enabled. If the function fails, it will return zero. Because it is possible that no features were enabled, a return value of zero does not always indicate an error. An application should call **FtpGetLastError** to determine if an error code has been set.

Remarks

The **FtpSetFeatures** function is used to enable a specific set of features for the current session. When a client connection is first established, features are enabled based on the server type and the server's response to the FEAT command. However, as the client issues commands to the server, if the server reports that the command is unrecognized that feature will automatically be disabled in the client. To enable or disable a specific feature, an application can use the **FtpEnableFeature** function.

For example, the first time an application calls the **FtpGetFileSize** function to determine the size of a file, the library will try to use the SIZE command. If the server reports that the SIZE command is not available, that feature will be disabled and the library will not use the command again during the session unless it is explicitly re-enabled. This is designed to prevent the library from repeatedly sending invalid commands to a server, which may result in the server aborting the connection.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

See Also

[FtpEnableFeature](#), [FtpGetFeatures](#)

FtpSetFileMode Function

```
INT WINAPI FtpSetFileMode(  
    HCLIENT hClient,  
    UINT nMode  
);
```

The **FtpSetFileMode** function sets the default file transfer mode for the current client session.

Parameters

hClient

Handle to the client session.

nMode

Specifies the default type of data transfer mode for files being opened or created on the server. This parameter can be one of the following values.

Value	Description
FILE_MODE_STREAM	The data is transmitted as a stream of bytes. This is the default client transfer mode.
FILE_MODE_BLOCK	The data is transmitted as a series of data blocks preceded by one or more header bytes. This transfer mode is currently not supported.
FILE_MODE_COMPRESSED	The data is transmitted in compressed form. This transfer mode is currently not supported.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The file transfer mode should be set before a file is opened or created on the server. Once the transfer mode is set, it is in effect for all files that are subsequently opened or created.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpOpenFile](#), [FtpSetFileStructure](#), [FtpSetFileType](#), [FtpSetPassiveMode](#)

FtpSetFileNameEncoding Function

```
INT WINAPI FtpSetFileNameEncoding(  
    HCLIENT hClient,  
    INT nEncoding  
);
```

The **FtpSetFileNameEncoding** function specifies what type of encoding will be used when file names are sent to the server.

Parameters

hClient

Handle to the client session.

nEncoding

An integer value which specifies the encoding type. It may be one of the following values:

Constant	Description
FTP_ENCODING_ANSI	File names are sent as 8-bit characters using the default character encoding for the current codepage. If the Unicode version of the functions are used, file names are converted from Unicode to ANSI using the current codepage before being sent to the server. This is the default encoding type.
FTP_ENCODING_UTF8	File names that contain non-ASCII characters are sent using UTF-8 encoding. This encoding type is only available on servers that advertise support for UTF-8 encoding and permit that encoding type to be enabled by the client.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpSetFileNameEncoding** function can be used to enable UTF-8 encoding of file names, which provides improved support for the use of international character sets. However, the server must provide support for UTF-8 encoding by advertising it in response to the FEAT command and it must support the OPTS command which is used to enable UTF-8 encoding. If the server does not advertise support for UTF-8, or the OPTS command fails with an error, then this function will fail with an error and the encoding type will not change.

Although it is possible to use the **FtpEnableFeature** function to explicitly enable the FTP_FEATURE_UTF8 feature, this is not recommended. If the server has not advertised support for UTF-8 encoding in response to the FEAT command, that typically indicates that UTF-8 encoding is not supported. Attempting to force UTF-8 encoding can result in unpredictable behavior when file names contain non-ASCII characters.

It is important to note that not all FTP servers support UTF-8 encoding, and in some cases servers which advertise support for UTF-8 encoding do not implement the feature correctly. For example, a server may allow a client to enable UTF-8 encoding, but once enabled will not permit the client to disable it. Some servers may advertise support for UTF-8 encoding, however if the underlying file system does not support UTF-8 encoded file names, any attempt to upload or download a file

may fail with an error indicating that the file cannot be found or created.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[FtpCommand](#), [FtpEnableFeature](#) [FtpGetFeatures](#), [FtpGetFileNameEncoding](#), [FtpSetFeatures](#)

FtpSetFilePermissions Function

```
INT WINAPI FtpSetFilePermissions(  
    HCLIENT hClient,  
    LPCTSTR lpszFileName,  
    DWORD dwPermissions  
);
```

The **FtpSetFilePermissions** function returns information about the access permissions for a specific file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

A pointer to a string which contains the name of the file to be updated. The filename cannot contain any wildcard characters.

dwPermissions

An unsigned integer which will specify the new access permissions for the file. The file permissions are represented as bit flags, and may be one or more of the following values combined with a bitwise Or operator:

Constant	Description
FILE_OWNER_READ	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.
FILE_OWNER_WRITE	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_OWNER_EXECUTE	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_GROUP_READ	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
FILE_GROUP_WRITE	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_GROUP_EXECUTE	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and

	search for files in that directory.
FILE_WORLD_READ	All users have permission to open the file for reading. This permission grants any user the right to download the file to the local system.
FILE_WORLD_WRITE	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.
FILE_WORLD_EXECUTE	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.

Return Value

If the function succeeds, the return value is a result code. If the function fails, the return value is `FTP_ERROR`. To get extended error information, call **FtpGetLastError**.

Remarks

This function uses the `SITE CHMOD` command to set the permissions for the file. This command is typically only supported on servers that are hosted on UNIX based systems. If the command is not supported, an error will be returned.

Users who are familiar with the UNIX operating system will recognize the **chmod** command used to change the file permissions. However, it should be noted that the numeric value used as an argument to the command is in octal, not decimal. For example, issuing the command **chmod 644 filename.txt** on a UNIX based system will make the file readable and writable by the owner, and readable by other users in the owner's group as well as all other users. The value 644 is an octal value, which is equivalent to the decimal value 420. If you were to mistakenly specify 644 as the value for the *dwPermissions* parameter, rather than the decimal value of 420, the permissions on the file would be incorrect. It is strongly recommended that you use the pre-defined constants to prevent this sort of error.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetFilePermissions](#), [FtpGetFileStatus](#)

FtpSetFileStructure Function

```
INT WINAPI FtpSetFileStructure(  
    HCLIENT hClient,  
    UINT nType  
);
```

The **FtpSetFileStructure** function sets the default file structure for the current client session, which indicates what type of file is being opened or created on the server.

Parameters

hClient

Handle to the client session.

nType

Specifies the default type of file structure being opened or created on the server. This parameter can be one of the following values.

Value	Description
FILE_STRUCT_NONE	The file has no inherent structure and is considered to be a stream of bytes. This is the default structure for file transfers.
FILE_STRUCT_RECORD	The file uses a record structure. This file structure is currently not supported.
FILE_STRUCT_PAGE	The file uses a page structure. This file structure is currently not supported.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The file structure should be set before a file is opened or created on the server. Once the file type is set, it is in effect for all files that are subsequently opened or created.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpOpenFile](#), [FtpSetFileMode](#), [FtpSetFileType](#), [FtpSetPassiveMode](#)

FtpSetFileTime Function

```
INT WINAPI FtpSetFileTime(  
    HCLIENT hClient,  
    LPCTSTR LpszFileName,  
    LPSYSTEMTIME lpFileTime  
);
```

The **FtpSetFileTime** function sets the modification time for the specified file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file.

lpFileTime

Points to a [SYSTEMTIME](#) structure that specifies the new modification time for the remote file.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is `FTP_ERROR`. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpSetFileTime** function will change the modification time of a file on the server. The values specified in the `SYSTEMTIME` structure are expected to represent UTC time, not time adjusted for the local system's timezone. If the values do represent the local time, it must be converted to UTC time prior to calling this function. To populate the `SYSTEMTIME` structure with the current time, use the **GetSystemTime** function.

When connected to an FTP server, this function uses the `MDTM` command to set the modification time for the specified file. Not all servers implement this command, in which case the function call will fail. Note that some servers only support the `MDTM` command to return, but not change, the file modification time.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetFileStatus](#), [FtpGetFileTime](#), [FtpOpenDirectory](#)

FtpSetFileType Function

```
INT WINAPI FtpSetFileType(  
    HCLIENT hClient,  
    UINT nFileType  
);
```

The **FtpSetFileType** function sets the default file type for the current client session, which indicates what type of file is being opened or created on the server.

Parameters

hClient

Handle to the client session.

nFileType

Specifies the default type of file being opened or created on the server. This parameter can be one of the following values.

Value	Description
FILE_TYPE_AUTO (0)	The file type should be automatically determined based on the file name extension. If the file extension is unknown, the file type should be determined based on the contents of the file. The library has an internal list of common text file extensions, and additional file extensions can be registered using the FtpRegisterFileType function.
FILE_TYPE_ASCII (1)	The file is a text file using the ASCII character set. For those servers which mark the end of a line with characters other than a carriage return and linefeed, it will be converted to the native client format. This is the file type used for directory listings.
FILE_TYPE_EBCDIC (2)	The file is a text file using the EBCDIC character set. Local files will be converted to EBCDIC when sent to the server. Remote files will be converted to the native ASCII character set when retrieved from the server. Not all servers support this file type. It is recommended that you only specify this type if you know that it is required by the server to transfer data correctly.
FILE_TYPE_IMAGE (3)	The file is a binary file and no data conversion of any type is performed on the file. This is the default file type for most data files and executable programs. If the type of file cannot be automatically determined, it will always be considered a binary file. If this file type is specified when uploading or downloading text files, the native end-of-line character sequences will be preserved.
FILE_TYPE_LOCAL (4)	The file is a binary file that uses the local byte size for the server platform. On most servers, this file type is considered to be the same as FILE_TYPE_IMAGE. Not all servers support this file type. It is recommended that you only specify this type if you know that it is required by the server to transfer data correctly.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return

value is `FTP_ERROR`. To get extended error information, call **FtpGetLastError**.

Remarks

The file type should be set before a file is opened or created on the server. Once the file type is set, it is in effect for all files that are subsequently opened or created. Some functions, such as **FtpOpenDirectory** and **FtpGetText**, will temporarily change the default file type to `FILE_TYPE_ASCII` and then restore the current file type when they return.

Calling this function has no practical effect when connected to an SFTP (SSH) server. They do not differentiate between text and binary files and the default file type will always be `FILE_TYPE_IMAGE`. If your application is uploading or downloading a text file, this difference between FTP and SFTP is important because the operating system that hosts the server may have different end-of-line character conventions than the client system. For example, if you download a text file from a UNIX system using SFTP, the end-of-line is indicated by a single linefeed (LF) character. However, on the Windows platform, the end-of-line is indicated by a carriage-return and linefeed sequence (CRLF).

The **FtpGetFileType** function can be used to determine the current file type.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

See Also

[FtpGetFileType](#), [FtpOpenFile](#), [FtpRegisterFileType](#), [FtpSetFileMode](#), [FtpSetFileStructure](#), [FtpSetPassiveMode](#)

FtpSetLastError Function

```
VOID WINAPI FtpSetLastError(  
    DWORD dwErrorCode  
);
```

The **FtpSetLastError** function sets the last error code for the current thread. This function is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or FTP_ERROR. Those functions which call **FtpSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **FtpGetLastError** function. The use of **FtpGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

See Also

[FtpGetErrorString](#), [FtpGetLastError](#)

FtpSetPassiveMode Function

```
INT WINAPI FtpSetPassiveMode(  
    HCLIENT hClient,  
    BOOL bPassiveMode  
);
```

The **FtpSetPassiveMode** function enables or disables passive mode file transfers for the specified client session.

Parameters

hClient

Handle to the client session.

bPassiveMode

A boolean flag which specifies that the client should enter passive mode and establish all connections with the server to transfer data.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

By default, the File Transfer Protocol uses active mode transfers, whereby the data connection is established from the server back to the local client. However, this can introduce problems for a client application that is behind a proxy server, firewall or a router which uses Network Address Translation (NAT). Enabling passive mode transfers instructs the client to create an outbound connection from the local system to the server for the data connection, similarly to how the control connection is established.

Not all servers may support passive mode, in which case an error will be returned to the client when this function is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpConnect](#), [FtpGetActivePorts](#), [FtpGetData](#), [FtpGetFile](#), [FtpProxyConnect](#), [FtpPutData](#), [FtpPutFile](#), [FtpSetActivePorts](#)

FtpSetPriority Function

```
INT WINAPI FtpSetPriority(  
    HCLIENT hClient,  
    INT nPriority  
);
```

The **FtpSetPriority** function specifies the priority for file transfers.

Parameters

hClient

Handle to the client session.

nPriority

An integer value which specifies the new priority for file transfers. It may be one of the following values:

Constant	Description
FTP_PRIORITY_NORMAL	The default priority which balances resource utilization and transfer speed. It is recommended that most applications use this priority.
FTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the transfer. It is typically used with worker threads running in the background when the amount of time required perform the transfer is not critical.
FTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the transfer and meters the bandwidth allocated for the transfer. This priority will increase the average amount of time required to complete a file transfer.
FTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the transfer, allocating more memory for internal buffering. It can be used when it is important to transfer the file quickly, and there are no other threads currently performing file transfers at the time.
FTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization while attempting to transfer the file as quickly as possible. If the file transfer is being performed in the main UI thread, this priority can cause the application to appear to become non-responsive. No events will be generated during the transfer.

Return Value

If the function succeeds, the return value is the previous file transfer priority. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpSetPriority** function can be used to control the processor usage, memory and network

bandwidth allocated for file transfers. The default priority balances resource utilization and transfer speed while ensuring that a single-threaded application remains responsive to the user. Lower priorities reduce the overall resource utilization at the expense of transfer speed. For example, if you create a worker thread to download a file in the background and want to ensure that it has a minimal impact on the process, the `FTP_PRIORITY_BACKGROUND` value can be used.

Higher priority values increase the memory allocated for the transfers and increases processor utilization for the transfer. The `FTP_PRIORITY_CRITICAL` priority maximizes transfer speed at the expense of system resources. It is not recommended that you increase the file transfer priority unless you understand the implications of doing so and have thoroughly tested your application. If the file transfer is being performed in the main UI thread, increasing the priority may interfere with the normal processing of Windows messages and cause the application to appear to become non-responsive. It is also important to note that when the priority is set to `FTP_PRIORITY_CRITICAL`, normal progress events will not be generated during the transfer.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetPriority](#)

FtpSetTimeout Function

```
INT WINAPI FtpSetTimeout(  
    HCLIENT hClient,  
    UINT nTimeout  
);
```

The **FtpSetTimeout** function sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is FTP_ERROR. To get extended error information, call **FtpGetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

See Also

[FtpConnect](#), [FtpGetTimeout](#), [FtpIsReadable](#), [FtpIsWritable](#), [FtpProxyConnect](#), [FtpRead](#), [FtpWrite](#)

FtpTaskAbort Function

```
BOOL WINAPI FtpTaskAbort(  
    UINT nTaskId,  
    DWORD dwMilliseconds  
);
```

Abort the specified asynchronous task.

Parameters

nTaskId

The task identifier.

dwMilliseconds

An unsigned integer that specifies the number of milliseconds to wait for the background task to abort.

Return Value

If the function succeeds and the worker thread has terminated, the return value is non-zero. A return value of zero indicates that the worker thread is still running or an error has occurred. To get extended error information, call the **FtpGetLastError** function.

Remarks

The **FtpTaskAbort** function signals the background worker thread associated with the task ID to abort the current operation and terminate as soon as possible. If the *dwMilliseconds* parameter has a value of zero, the function returns immediately after the background thread has been signaled. If the *dwMilliseconds* parameter is non-zero, the function will wait that amount of time for the background thread to terminate.

This function should never be called from within the event handler for an asynchronous task because it can cause the process to deadlock. To abort a file transfer within an event handler, use the **FtpCancel** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

See Also

[FtpTaskDone](#), [FtpTaskResume](#), [FtpTaskSuspend](#), [FtpTaskWait](#)

FtpTaskDone Function

```
BOOL WINAPI FtpTaskDone(  
    UINT nTaskId  
);
```

Determine if an asynchronous task has completed.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has completed, this function returns a non-zero value. A return value of zero indicates that the worker thread is still running or an error has occurred. To get extended error information, call the **FtpGetLastError** function.

Remarks

The **FtpTaskDone** function is used to determine if the specified asynchronous task has completed. If you use this function to poll the status of a background task from within the main UI thread, you must ensure that Windows messages are processed so that the application remains responsive to the end-user. To check if a background transfer has completed, it is recommended that you use a timer to periodically call this function rather than calling it repeatedly within a loop.

To determine if the task completed successfully, the **FtpGetTaskError** function will return the last error code associated with the task. A return value of zero indicates success, while a non-zero return value specifies an error code that indicates the cause of the failure. The last error code for the task can also be retrieved using the **FtpTaskWait** function, which causes the application to wait for the asynchronous task to complete.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[FtpGetTaskError](#), [FtpTaskAbort](#), [FtpTaskResume](#), [FtpTaskSuspend](#), [FtpTaskWait](#)

FtpTaskResume Function

```
BOOL WINAPI FtpTaskResume(  
    UINT nTaskId  
);
```

Resume execution of an asynchronous task.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has resumed, this function returns a non-zero value. A return value of zero indicates that an error has occurred. To get extended error information, call the **FtpGetLastError** function.

Remarks

The **FtpTaskResume** function resumes execution of the background worker thread that was previously suspended using the **FtpTaskSuspend** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[FtpTaskAbort](#), [FtpTaskDone](#), [FtpTaskSuspend](#), [FtpTaskWait](#)

FtpTaskSuspend Function

```
BOOL WINAPI FtpTaskSuspend(  
    UINT nTaskId  
);
```

Suspend execution of an asynchronous task.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has resumed, this function returns a non-zero value. A return value of zero indicates that an error has occurred. To get extended error information, call the **FtpGetLastError** function.

Remarks

The **FtpTaskSuspend** function will suspend execution of the background worker thread associated with the task. Once the task has been suspended, it will no longer be scheduled for execution, however the client session will remain active and the task may be resumed using the **FtpTaskResume** function. Note that if a task is suspended for a long period of time, the background operation may fail because it has exceeded the timeout period imposed by the server.

This function should never be called from within the event handler for an asynchronous task because it can cause the process to deadlock.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[FtpTaskAbort](#), [FtpTaskDone](#), [FtpTaskResume](#), [FtpTaskWait](#)

FtpTaskWait Function

```
BOOL WINAPI FtpTaskWait(  
    UINT nTaskId,  
    DWORD dwMilliseconds,  
    DWORD dwReserved,  
    LPDWORD lpdwElapsed,  
    LPDWORD lpdwError  
);
```

Wait for an asynchronous task to complete.

Parameters

nTaskId

The task identifier.

dwMilliseconds

An unsigned integer that specifies the number of milliseconds to wait for the background task to complete.

dwReserved

An unsigned integer reserved for future use. This value should always be zero.

lpdwElapsed

A pointer to an unsigned integer that will contain elapsed time in milliseconds when the function returns. If this information is not required, this parameter may be NULL.

lpdwError

A pointer to an unsigned integer that will contain the error code associated with the completed task. If this information is not required, this parameter may be NULL.

Return Value

If the function succeeds and the worker thread has terminated, the return value is non-zero. A return value of zero indicates that the worker thread is still running or an error has occurred. To get extended error information, call the **FtpGetLastError** function.

Remarks

The **FtpTaskWait** function waits for the specified task to complete. If the task is active and the *dwMilliseconds* parameter is non-zero, this function will cause the current working thread to block until the task completes or the amount of time exceeds the number of milliseconds specified by the caller. If the *dwMilliseconds* parameter is zero, then this function will poll the status of the task and return immediately to the caller.

If the specified task has already completed at the time this function is called, the function will return immediately without causing the current thread to block. If the *lpdwElapsed* parameter is not NULL, it will contain the number of milliseconds that it took for the task to complete. If the *lpdwError* parameter is not NULL, it will contain the last error code value that was set by the worker thread before it terminated. If this value is zero, that means that the background operation was successful and no error occurred. A non-zero value will indicate that the background operation has failed.

You should not call this function from the main UI thread with a long timeout period to wait for a background task to complete. Windows messages will not be processed while this function is blocked waiting for the background task to complete, and this can cause your application to

appear non-responsive to the end-user. If you have a GUI application and you need to periodically check to see if a task has completed, create a timer to periodically call the **FtpTaskDone** function. When it returns a non-zero value (indicating that the task has completed), you can safely call **FtpTaskWait** to obtain the elapsed time and last error code without blocking the current thread.

This function should never be called from within the event handler for an asynchronous task because it can cause the process to deadlock.

Example

```
UINT nTaskId;

// Begin a file transfer in the background

nTaskId = FtpAsyncGetFile(hClient,
                        lpszLocalFile,
                        lpszRemoteFile,
                        FTP_TRANSFER_DEFAULT,
                        0,
                        NULL,
                        0);

if (nTaskId != 0)
{
    DWORD dwError = NO_ERROR;
    DWORD dwElapsed = 0;

    // Wait for the transfer to complete
    FtpTaskWait(nTaskId, INFINITE, &dwElapsed, &dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

See Also

[FtpTaskDone](#), [FtpTaskResume](#), [FtpTaskSuspend](#), [FtpTaskWait](#)

FtpUninitialize Function

```
VOID WINAPI FtpUninitialize();
```

The **FtpUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **FtpInitialize** call before it can call any of the other library functions. When it has completed the use of library, the application must call **FtpUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There must be a call to **FtpUninitialize** for every successful call to **FtpInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpAsyncConnect](#), [FtpAsyncProxyConnect](#), [FtpConnect](#), [FtpDisconnect](#), [FtpInitialize](#),
[FtpProxyConnect](#), [FtpReset](#)

FtpUploadFile Function

```
BOOL WINAPI FtpUploadFile(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszFileURL,  
    UINT nTimeout  
    DWORD dwOptions  
    LPFTPTRANSFERSTATUS lpStatus  
    FTPEVENTPROC lpEventProc  
    DWORD_PTR dwParam  
);
```

The **FtpUploadFile** function uploads the specified file from the local system to the server.

Parameters

lpszLocalFile

A pointer to a string that specifies the file on the local system that will be uploaded to the server. The file pathing and name conventions must be that of the local host.

lpszFileURL

A pointer to a string that specifies the complete URL of the file that will be created or overwritten on the server. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
FTP_OPTION_PASSIVE	This option specifies the client should attempt to establish the data connection with the server. When the client uploads or downloads a file, normally the server establishes a second connection back to the client which is used to transfer the file data. However, if the local system is behind a firewall or a NAT router, the server may not be able to create the data connection and the transfer will fail. By specifying this option, it forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it

	returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should use the AUTH command to negotiate an explicit secure connection. Some servers may only require this when connecting to the server on ports other than 990.

lpStatus

A pointer to an FTPTRANSFERSTATUS structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpUploadFile** function provides a convenient way for an application to upload a file in a single function call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session, change the current working directory if necessary and then upload the file to the server. The URL must be complete, and specify either a standard or secure FTP scheme:

```
[ftp|ftps|sftp]://[username : password] @[remotehost] [:remoteport] /
[path / ...] [filename] [;type=a|i]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. If a path is specified as part of the URL, the function will attempt to change the current working directory. Note that the path in an FTP URL is relative to the home directory of the user account and is not an absolute path starting at the root directory on the server. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "ftp" scheme is used and the FTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, either the "ftps" or "sftp" scheme must be specified.

The optional "type" value at the end of the file name determines if the file should be uploaded as a text or binary file. A value of "a" specifies that the file should be uploaded as a text file. A value of "i" specifies that the file should be uploaded as a binary file. If the type is not explicitly specified, the file will be uploaded as a binary file.

The *lpStatus* parameter can be used by the application to determine the final status of the

transfer, including the total number of bytes copied, the amount of time elapsed and other information related to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The *lpEventProc* parameter specifies a pointer to a function which will be periodically called during the file transfer process. This can be used to check the status of the transfer by calling **FtpGetTransferStatus** and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The *dwParam* parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the *this* pointer as the value, and then cast it back to an object pointer inside the callback function. If no event handler is required, then a NULL pointer can be specified as the value for *lpEventProc* and the *dwParam* parameter will be ignored.

The **FtpUploadFile** function is designed to provide a simpler interface for uploading a file. However, complex connections such as those using a proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **FtpConnect** and then use **FtpPutFile** to upload the file.

Example

```
FTPTRANSFERSTATUS ftpStatus;
LPCTSTR lpszLocalFile = _T("c:\\temp\\database.mdb");
LPCTSTR lpszFileURL =
_T("ftp://update:secret@ftp.example.com/updates/database.mdb");
BOOL bResult;

// Upload the file using the specified URL
bResult = FtpUploadFile(lpszLocalFile,
                        lpszFileURL,
                        FTP_OPTION_PASSIVE,
                        &ftpStatus,
                        NULL, 0);

if (!bResult)
{
    TCHAR szError[128];

    // Display a message box that describes the error
    FtpGetErrorString(FtpGetLastError(), szError, 128);
    MessageBox(NULL, szError, NULL, MB_ICONEXCLAMATION|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpEventProc](#), [FtpDownloadFile](#), [FtpGetTransferStatus](#), [FtpPutFile](#), [FTPTRANSFERSTATUS](#)

FtpUploadFileEx Function

```
BOOL WINAPI FtpUploadFileEx(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszFileURL,  
    UINT nTimeout  
    DWORD dwOptions  
    LPFTPTRANSFERSTATUSEX lpStatus  
    FTPEVENTPROC lpEventProc  
    DWORD_PTR dwParam  
);
```

The **FtpUploadFileEx** function uploads the specified file from the local system to the server.

Parameters

lpszLocalFile

A pointer to a string that specifies the file on the local system that will be uploaded to the server. The file pathing and name conventions must be that of the local host.

lpszFileURL

A pointer to a string that specifies the complete URL of the file that will be created or overwritten on the server. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
FTP_OPTION_PASSIVE	This option specifies the client should attempt to establish the data connection with the server. When the client uploads or downloads a file, normally the server establishes a second connection back to the client which is used to transfer the file data. However, if the local system is behind a firewall or a NAT router, the server may not be able to create the data connection and the transfer will fail. By specifying this option, it forces the client to establish an outbound data connection with the server. It is recommended that applications use passive mode whenever possible.
FTP_OPTION_FIREWALL	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it

	returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
FTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.
FTP_OPTION_SECURE_EXPLICIT	This option specifies the client should use the AUTH command to negotiate an explicit secure connection. Some servers may only require this when connecting to the server on ports other than 990.

lpStatus

A pointer to an FTPTRANSFERSTATUSEX structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **FtpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpUploadFileEx** function provides a convenient way for an application to upload a file in a single function call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session, change the current working directory if necessary and then upload the file to the server. The URL must be complete, and specify either a standard or secure FTP scheme:

```
[ftp|ftps|sftp]://[username : password] @] remotehost [:remoteport] /
[path / ...] [filename] [;type=a|i]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. If a path is specified as part of the URL, the function will attempt to change the current working directory. Note that the path in an FTP URL is relative to the home directory of the user account and is not an absolute path starting at the root directory on the server. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "ftp" scheme is used and the FTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, either the "ftps" or "sftp" scheme must be specified.

The optional "type" value at the end of the file name determines if the file should be uploaded as a text or binary file. A value of "a" specifies that the file should be uploaded as a text file. A value of "i" specifies that the file should be uploaded as a binary file. If the type is not explicitly specified, the file will be uploaded as a binary file.

The *lpStatus* parameter can be used by the application to determine the final status of the

transfer, including the total number of bytes copied, the amount of time elapsed and other information related to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The *lpEventProc* parameter specifies a pointer to a function which will be periodically called during the file transfer process. This can be used to check the status of the transfer by calling **FtpGetTransferStatus** and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The *dwParam* parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the *this* pointer as the value, and then cast it back to an object pointer inside the callback function. If no event handler is required, then a NULL pointer can be specified as the value for *lpEventProc* and the *dwParam* parameter will be ignored.

The **FtpUploadFileEx** function is designed to provide a simpler interface for uploading a file. However, complex connections such as those using a proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **FtpConnect** and then use **FtpPutFileEx** to upload the file.

Example

```
FTPTRANSFERSTATUSEX ftpStatus;
LPCTSTR lpszLocalFile = _T("c:\\temp\\database.mdb");
LPCTSTR lpszFileURL =
_T("ftp://update:secret@ftp.example.com/updates/database.mdb");
BOOL bResult;

// Upload the file using the specified URL
bResult = FtpUploadFileEx(lpszLocalFile,
                          lpszFileURL,
                          FTP_OPTION_PASSIVE,
                          &ftpStatus,
                          NULL, 0);

if (!bResult)
{
    TCHAR szError[128];

    // Display a message box that describes the error
    FtpGetErrorString(FtpGetLastError(), szError, 128);
    MessageBox(NULL, szError, NULL, MB_ICONEXCLAMATION|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpEventProc](#), [FtpDownloadFileEx](#), [FtpGetTransferStatusEx](#), [FtpPutFileEx](#), [FTPTRANSFERSTATUSEX](#)

FtpValidateUrl Function

```
BOOL WINAPI FtpValidateUrl(  
    LPCTSTR LpszURL  
);
```

The **FtpValidateUrl** function determines if a string represents a valid FTP URL.

Parameters

LpszURL

A pointer to a string that specifies the URL to validate.

Return Value

If the specified URL is valid and the host name can be resolved to an IP address, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpValidateUrl** function will check the value of a string to ensure that it represents a complete, valid URL using either a standard or secure FTP scheme. This function will not establish a connection with the server to verify that it exists, it will only attempt to resolve the host name to an IP address. If the remote host is specified as an IP address, this function will check to make sure that the address is formatted correctly. Note that if you wish to specify an IPv6 address, you must enclose the address in brackets.

To establish a connection with a server using a URL, use the **FtpConnectUrl** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpConnectUrl](#)

FtpVerifyFile Function

```
BOOL WINAPI FtpVerifyFile(  
    HCLIENT hClient,  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszRemoteFile,  
    DWORD dwOptions  
);
```

The **FtpVerifyFile** function attempts to verify that the contents of a file on the local system are the same as the specified file on the server.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A pointer to a string that specifies the name of file on the local system.

lpszRemoteFile

A pointer to a string that specifies the name of the file on the server.

dwOptions

Specifies the options that may be used when comparing the files. This parameter may be one or more of the following values:

Value	Description
FTP_VERIFY_DEFAULT	File verification should use the best option available based on the available server features. If the server supports the XMD5 command, the library will calculate an MD5 hash of the local file contents and compare the value with the file on the server. If the server does not support the XMD5 command, but it does support the XCRC command, the library will calculate a CRC32 checksum of the local file contents and compare the value with the file on the server. If the server does not support either the XMD5 or XCRC commands, the library will compare the size of the local and remote files.
FTP_VERIFY_SIZE	Files are verified by comparing the number of bytes of data in the local and remote files. This is the least reliable method, and should only be used if the server does not support either the XMD5 or XCRC commands.
FTP_VERIFY_CRC32	Files are verified by calculating a CRC32 checksum of the local file contents and comparing it with the value returned by the server in response to the XCRC command. This method should only be used if the server does not support the XMD5 command.
FTP_VERIFY_MD5	Files are verified by calculating an MD5 hash of the local file contents and comparing it with the value returned by the server in response to the XMD5 command. This is the preferred method for performing file verification.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetLastError**.

Remarks

The **FtpVerifyFile** function will attempt to verify that the contents of the local and remote files are identical using one of several methods, based on the features that the server supports. Preference will be given to the most reliable method available, using either an MD5 hash, a CRC-32 checksum or comparing the size of the file, in that order.

It is not recommended that you use this function with text files because of the different end-of-line conventions used by different operating systems. For example, a text file on a Windows system uses a carriage-return and linefeed pair to indicate the end of a line of text. However, on a UNIX system, a single linefeed is used to indicate the end of a line. This can cause the **FtpVerifyFile** function to indicate the files are not identical, even though the only difference is in the end-of-line characters that are used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpDeleteFile](#), [FtpGetFile](#), [FtpPutFile](#)

FtpWrite Function

```
INT WINAPI FtpWrite(  
    HCLIENT hClient,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **FtpWrite** function sends the specified number of bytes to the server.

Parameters

hClient

Handle to the client session.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is `FTP_ERROR`. To get extended error information, call **FtpGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the next asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpEnableEvents](#), [FtpIsBlocking](#), [FtpIsReadable](#), [FtpIsWritable](#), [FtpPutData](#), [FtpPutFile](#), [FtpRead](#), [FtpRegisterEvent](#)

File Transfer Protocol Data Structures

- FTPCLIENTQUOTA
- FTPFILESTATUS
- FTPFILESTATUSEX
- FTPTRANSFERSTATUS
- FTPTRANSFERSTATUSEX
- INITDATA
- SECURITYCREDENTIALS
- SECURITYINFO
- SYSTEMTIME

FTPCLIENTQUOTA Structure

This structure is used by the [FtpGetClientQuota](#) function to return information about the file quota for the current client session.

```
typedef struct _FTPCLIENTQUOTA
{
    DWORD    dwFileCount;
    DWORD    dwFileLimit;
    DWORD    dwDiskUsage;
    DWORD    dwDiskLimit;
} FTPCLIENTQUOTA, *LPFTPCLIENTQUOTA;
```

Members

dwFileCount

An unsigned integer value which specifies the number of files that the user has created. If file quotas have not been enabled for the current user, this value will be zero.

dwFileLimit

An unsigned integer value which specifies the maximum number of files that may be created by the user. If file quotas have not been enabled for the current user, this value will be zero.

dwDiskUsage

An unsigned integer value which specifies the number of bytes of disk storage that has been allocated by the current user. If file quotas have not been enabled for the current user, this value will be zero.

dwDiskLimit

An unsigned integer value which specifies the maximum number of bytes of disk storage that may be allocated by the current user. If file quotas have not been enabled for the current user, this value will be zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftpv10.lib

FTPFILESTATUS Structure

This structure is used by the **FtpEnumFiles**, **FtpGetFirstFile** and **FtpGetNextFile** functions to return information about files on the server.

```
typedef struct _FTPFILESTATUS
{
    TCHAR          szFileName[FTP_MAXFILENAMELEN];
    TCHAR          szFileOwner[FTP_MAXOWNERNAMELEN];
    TCHAR          szFileGroup[FTP_MAXGROUPNAMELEN];
    BOOL           bIsDirectory;
    DWORD          dwFileSize;
    DWORD          dwFileLinks;
    DWORD          dwFileVersion;
    DWORD          dwFilePerms;
    SYSTEMTIME     stFileDate;
} FTPFILESTATUS, *LPFTPFILESTATUS;
```

Members

szFileName

A string buffer which contains the name of the file on the server.

szFileOwner

A string buffer which contains the name of the user that owns the file on the server. Note that not all server types support the concept of file ownership by a user. Some UNIX systems will not provide this information if an anonymous login was used. For the proprietary Sterling directory formats, the "mailbox" is stored in this member.

szFileGroup

A string buffer which contains the name of the group that owns the file on the server. Note that not all server types support the concept of file ownership by a group. For the proprietary Sterling directory formats, the "batch number" is stored in this member, with the character # prepended for the format FTP_DIRECTORY_STERLING_2.

bIsDirectory

A boolean flag which specifies if the file is actually a subdirectory.

dwFileSize

The size of the file in bytes on the server. Servers that return file information in an MS-DOS format will always set this value to zero if the file refers to a subdirectory. If the file is a text file, the file size on the server may be different than the size on the local host if different end-of-line character conventions are used. It should be noted that under VMS, the file size is reported in 512 byte blocks, so the size should be considered approximate on that platform.

dwFileLinks

The number of links to the file. Note that not all server types support the concept of file links, in which case this value will be zero.

dwFileVersion

The number of revisions made to the file. Note that not all server types support the concept of file versioning, in which case this value will be zero. Currently this value will only be non-zero for VMS platforms.

dwFilePerms

The permissions associated with the file. This value is actually a combination of bits that specify

the individual permissions for the file owner, group and world (all other users). For those familiar with UNIX, the file permissions are the same as those used by the `chmod` command. For the proprietary Sterling directory formats, a bit map representing the status codes and transfer protocol of the file is stored in this member.

stFileDate

A `SYSTEMTIME` structure which specifies the date that the file was created or last modified.

File Permissions

Constant	Description
<code>FILE_OWNER_READ</code>	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.
<code>FILE_OWNER_WRITE</code>	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
<code>FILE_OWNER_EXECUTE</code>	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
<code>FILE_GROUP_READ</code>	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
<code>FILE_GROUP_WRITE</code>	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
<code>FILE_GROUP_EXECUTE</code>	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
<code>FILE_WORLD_READ</code>	All users have permission to open the file for reading. This permission grants any user the right to download the file to the local system.
<code>FILE_WORLD_WRITE</code>	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.
<code>FILE_WORLD_EXECUTE</code>	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.

Sterling Status Codes

Bits 0-25 correspond to letters of the alphabet, most of which have distinct meanings in the Sterling formats.

Letter code	Bit position	Hexadecimal value
-------------	--------------	-------------------

A	0	1h
B	1	2h
C	2	4h
<i>n-th letter of alphabet</i>	n-1	2 to the (n-1) power
Z	25	2000000h

For the proprietary Sterling directory formats, bits 26-31 represent the transfer protocol associated with the file:

Protocol	Bit position	Hexadecimal value	Constant
TCP	26	4000000h	FTP_STERLING_STATUS_TCP
FTP	27	8000000h	FTP_STERLING_STATUS_FTP
BSC	28	10000000h	FTP_STERLING_STATUS_BSC
ASC	29	20000000h	FTP_STERLING_STATUS_ASC
FTS	30	40000000h	FTP_STERLING_STATUS_FTS
other	31	80000000h	FTP_STERLING_STATUS_OTHER

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

FTPFILESTATUSEX Structure

This structure is used by the **FtpEnumFilesEx**, **FtpGetFirstFileEx** and **FtpGetNextFileEx** functions to return information about files on the server. This structure is designed for use with extended functions that support files larger than 4GB.

```
typedef struct _FTPFILESTATUSEX
{
    TCHAR          szFileName[FTP_MAXFILENAMELEN];
    TCHAR          szFileOwner[FTP_MAXOWNERNAMELEN];
    TCHAR          szFileGroup[FTP_MAXGROUPNAMELEN];
    BOOL           bIsDirectory;
    ULARGE_INTEGER uiFileSize;
    DWORD          dwFileLinks;
    DWORD          dwFileVersion;
    DWORD          dwFilePerms;
    DWORD          dwFileFlags;
    SYSTEMTIME     stFileDate;
} FTPFILESTATUSEX, *LPFTPFILESTATUSEX;
```

Members

szFileName

A string buffer which contains the name of the file on the server.

szFileOwner

A string buffer which contains the name of the user that owns the file on the server. Note that not all server types support the concept of file ownership by a user. Some UNIX systems will not provide this information if an anonymous login was used. For the proprietary Sterling directory formats, the "mailbox" is stored in this member.

szFileGroup

A string buffer which contains the name of the group that owns the file on the server. Note that not all server types support the concept of file ownership by a group. For the proprietary Sterling directory formats, the "batch number" is stored in this member, with the character # prepended for the format FTP_DIRECTORY_STERLING_2.

bIsDirectory

A boolean flag which specifies if the file is actually a subdirectory.

uiFileSize

The size of the file in bytes on the server. Servers that return file information in an MS-DOS format will always set this value to zero if the file refers to a subdirectory. If the file is a text file, the file size on the server may be different than the size on the local host if different end-of-line character conventions are used. It should be noted that under VMS, the file size is reported in 512 byte blocks, so the size should be considered approximate on that platform.

dwFileLinks

The number of links to the file. Note that not all server types support the concept of file links, in which case this value will be zero.

dwFileVersion

The number of revisions made to the file. Note that not all server types support the concept of file versioning, in which case this value will be zero. Currently this value will only be non-zero for VMS platforms.

dwFilePerms

The permissions associated with the file. This value is actually a combination of bits that specify the individual permissions for the file owner, group and world (all other users). For those familiar with UNIX, the file permissions are the same as those used by the `chmod` command. For the proprietary Sterling directory formats, a bit map representing the status codes and transfer protocol of the file is stored in this member.

dwFileFlags

This structure member is reserved for future use.

stFileDate

A [SYSTEMTIME](#) structure which specifies the date that the file was created or last modified.

File Permissions

Constant	Description
FILE_OWNER_READ	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.
FILE_OWNER_WRITE	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_OWNER_EXECUTE	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_GROUP_READ	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
FILE_GROUP_WRITE	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
FILE_GROUP_EXECUTE	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
FILE_WORLD_READ	All users have permission to open the file for reading. This permission grants any user the right to download the file to the local system.
FILE_WORLD_WRITE	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.
FILE_WORLD_EXECUTE	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.

Sterling Status Codes

Bits 0-25 correspond to letters of the alphabet, most of which have distinct meanings in the Sterling formats.

Letter code	Bit position	Hexadecimal value
A	0	1h
B	1	2h
C	2	4h
<i>n-th letter of alphabet</i>	n-1	2 to the (n-1) power
Z	25	2000000h

For the proprietary Sterling directory formats, bits 26-31 represent the transfer protocol associated with the file:

Protocol	Bit position	Hexadecimal value	Constant
TCP	26	4000000h	FTP_STERLING_STATUS_TCP
FTP	27	8000000h	FTP_STERLING_STATUS_FTP
BSC	28	10000000h	FTP_STERLING_STATUS_BSC
ASC	29	20000000h	FTP_STERLING_STATUS_ASC
FTS	30	40000000h	FTP_STERLING_STATUS_FTS
other	31	80000000h	FTP_STERLING_STATUS_OTHER

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

FTPTRANSFERSTATUS Structure

This structure is used by the **FtpGetTransferStatus** function to return information about a file transfer in progress.

```
typedef struct _FTPTRANSFERSTATUS
{
    DWORD    dwBytesTotal;
    DWORD    dwBytesCopied;
    DWORD    dwBytesPerSecond;
    DWORD    dwTimeElapsed;
    DWORD    dwTimeEstimated;
    TCHAR    szLocalFile[FTP_MAXFILENAMELEN];
    TCHAR    szRemoteFile[FTP_MAXFILENAMELEN];
} FTPTRANSFERSTATUS, *LPFTPTRANSFERSTATUS;
```

Members

dwBytesTotal

The total number of bytes that will be transferred. If the file is being copied from the server to the local host, this is the size of the remote file. If the file is being copied from the local host to the server, it is the size of the local file. If the file size cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the file transfer is completed. This is based on the average number of bytes transferred per second.

szLocalFile

A pointer to a string which specifies the local file that is being copied to or from the server.

szRemoteFile

A pointer to a string which specifies the remote file that is being copied to or from the local system.

Remarks

If the option `FTP_OPTION_HIRES_TIMER` has been specified when connecting to the server, the values returned in the *dwTimeElapsed* and *dwTimeEstimated* members will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small files over a fast network connection.

If you are uploading or downloading large files which exceed 4GB, you should use the **FTPTRANSFERSTATUSEX** structure which uses 64-bit integers for the file size.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

See Also

[FtpGetTransferStatus](#), [FtpGetTransferStatusEx](#), [FTPTRANSFERSTATUSEX](#)

FTPTRANSFERSTATUSEX Structure

This structure is used by the **FtpGetTransferStatusEx** function to return information about a file transfer in progress. This structure is designed for use with extended functions that support files larger than 4GB.

```
typedef struct _FTPTRANSFERSTATUSEX
{
    ULARGE_INTEGER dwBytesTotal;
    ULARGE_INTEGER dwBytesCopied;
    DWORD          dwBytesPerSecond;
    DWORD          dwTimeElapsed;
    DWORD          dwTimeEstimated;
    DWORD          dwReserved;
    TCHAR          szLocalFile[FTP_MAXFILENAMELEN];
    TCHAR          szRemoteFile[FTP_MAXFILENAMELEN];
} FTPTRANSFERSTATUSEX, *LPFTPTRANSFERSTATUSEX;
```

Members

uiBytesTotal

The total number of bytes that will be transferred. If the file is being copied from the server to the local host, this is the size of the remote file. If the file is being copied from the local host to the server, it is the size of the local file. If the file size cannot be determined, this value will be zero.

uiBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the file transfer is completed. This is based on the average number of bytes transferred per second.

dwReserved

This structure member is reserved for future use.

szLocalFile

A pointer to a string which specifies the local file that is being copied to or from the server.

szRemoteFile

A pointer to a string which specifies the remote file that is being copied to or from the local system.

Remarks

If the option `FTP_OPTION_HIRES_TIMER` has been specified when connecting to the server, the values returned in the *dwTimeElapsed* and *dwTimeEstimated* members will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small files over a fast network connection.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

See Also

[FtpGetTransferStatus](#), [FtpGetTransferStatusEx](#), [FTPTRANSFERSTATUS](#)

INITDATA Structure

This structure contains information about the SocketTools library when the initialization function returns.

```
typedef struct _INITDATA
{
    DWORD        dwSize;
    DWORD        dwVersionMajor;
    DWORD        dwVersionMinor;
    DWORD        dwVersionBuild;
    DWORD        dwOptions;
    DWORD_PTR    dwReserved1;
    DWORD_PTR    dwReserved2;
    TCHAR        szDescription[128];
} INITDATA, *LPINITDATA;
```

Members

dwSize

Size of this structure. This structure member must be set to the size of the structure prior to calling the initialization function. Failure to do so will cause the function to fail.

dwVersionMajor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the major version number for the library.

dwVersionMinor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the minor version number for the library.

dwVersionBuild

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the build number for the library.

dwOptions

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved1

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved2

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

szDescription

A null terminated string which describes the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

SECURITYCREDENTIALS Structure

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on

	<p>what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.</p>
SECURITY_PROTOCOL_TLS10	<p>The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS11	<p>The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS12	<p>The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.</p>
SECURITY_PROTOCOL_SSH	<p>Either version 1.0 or 2.0 of the Secure Shell protocol should be used when establishing the connection. The correct protocol is automatically selected based on the version of the protocol that is supported by the server.</p>
SECURITY_PROTOCOL_SSH1	<p>The Secure Shell 1.0 protocol should be used when establishing the connection. This is an older version of the protocol which should not be used unless explicitly required by the server. Most modern SSH server support version 2.0 of the protocol.</p>
SECURITY_PROTOCOL_SSH2	<p>The Secure Shell 2.0 protocol should be used when establishing the connection. This is the default version of the protocol that is supported by most SSH servers.</p>

dwOptions

A value which specifies one or options. This value should always be zero for connections using SSH. This member is constructed by using a bitwise operator with any of the following values:



Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A pointer to a null terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A pointer to a null terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A pointer to a null terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A pointer to a null terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
------------	-------------

CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
----	---

MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
----	--

ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are
------	---

installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

A pointer to a null terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used for SSH connections and should always be NULL when establishing a secure connection using SSL or TLS.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* member should be set to CREDENTIAL_STORE_FILENAME and the *lpszCertStore* member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS#12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the *lpszPassword* member or the library will be unable to access the certificate.

Note that the *lpszUserName* and *lpszPassword* members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

The TLS 1.1 and TLS 1.2 protocols are only supported on Windows 7, Windows Server 2008 R2 and later versions of the platform. If these options are specified and the application is running on Windows XP or Windows Vista, the protocol version will be downgraded to TLS 1.0 for backwards compatibility with those platforms.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

SECURITYINFO Structure

This structure contains information about a secure connection that has been established with a server.

```
typedef struct _SECURITYINFO
{
    DWORD        dwSize;
    DWORD        dwProtocol;
    DWORD        dwCipher;
    DWORD        dwCipherStrength;
    DWORD        dwHash;
    DWORD        dwHashStrength;
    DWORD        dwKeyExchange;
    DWORD        dwCertStatus;
    SYSTEMTIME   stCertIssued;
    SYSTEMTIME   stCertExpires;
    LPCTSTR      lpszCertIssuer;
    LPCTSTR      lpszCertSubject;
    LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to **sizeof(SECURITYINFO)** prior to passing the address of this structure to the function. Note that if this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the protocol that was selected to establish the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be

	<p>used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.</p>
SECURITY_PROTOCOL_TLS10	<p>The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS11	<p>The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS12	<p>The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.</p>
SECURITY_PROTOCOL_SSH1	<p>The Secure Shell 1.0 protocol has been selected. This protocol has been deprecated and is no longer widely used. It is not recommended that this protocol be used when establishing secure connections. This protocol can only be specified when connecting to an SSH server and is not supported with any other application protocol.</p>
SECURITY_PROTOCOL_SSH2	<p>The Secure Shell 2.0 protocol has been selected. This is the most commonly used version of the protocol. It is recommended that this version of the protocol be used unless the server explicitly requires the client to use an earlier version. This protocol can only be specified when connecting to an SSH server and is not supported with any other application protocol.</p>

A numeric value which specifies the cipher that was selected when establishing the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_CIPHER_RC2	The RC2 block cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
SECURITY_CIPHER_DES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher (also known as the Rijndael cipher) is a fixed block size cipher which use a key size of 128, 192 or 256 bits. This cipher is supported on Windows XP SP3 and later versions of the operating system.
SECURITY_CIPHER_SKIPJACK	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
SECURITY_CIPHER_BLOWFISH	The Blowfish block cipher was selected. This is a variable key length cipher up to 448 bits, using a 64-bit block size.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected. One of the following values will be returned:

Constant	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.

SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected. One of the following values will be returned:

Constant	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography. This key exchange algorithm is only supported on Windows XP SP3 and later versions of the operating system.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server. This member only has meaning for connections using the SSL or TLS protocols. One of the following values will be returned:

Constant	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertIssuer

A pointer to a string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertSubject

A pointer to a string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszFingerprint

A pointer to a string which contains a sequence of hexadecimal values that uniquely identify the server. This member is only used when a connection has been established using the Secure Shell (SSH) protocol.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

SYSTEMTIME Structure

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include windows.h.

File Transfer Protocol Server Library

Implements a server that enables the application to send and receive files using the File Transfer Protocol.

Reference

- [Functions](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

File Name	CSFTSV10.DLL
Version	10.0.1468.2518
LibID	5CED7337-69F7-4662-B173-42FA4EEB64E3
Import Library	CSFTSV10.LIB
Dependencies	None
Standards	RFC 959, RFC 1579, RFC 2228

Overview

This library provides an interface for implementing an embedded, lightweight server that can be used to exchange files with a client using the standard File Transfer Protocol. The server can accept connections from any third-party application or a program developed using the SocketTools FTP client API.

The application specifies an initial server configuration and then responds to events that are raised by the API when the client sends a request to the server. An application may implement only minimal handlers for most events, in which case the default actions are performed for most standard FTP commands. However, an application may also use the event mechanism to filter specific commands or to extend the protocol by providing custom implementations of existing commands or add entirely new commands.

The server supports active and passive mode file transfers, has compatibility options for NAT router and firewall support, and provides support for secure file transfers using explicit TLS. Secure connections require a valid security certificate to be installed on the system.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This library provides an implementation of a multithreaded server which should only be used with

languages that support the creation of multithreaded applications. It is important that you do not attempt to link against static libraries which were not built with support for threading.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

File Transfer Protocol Server Functions

Function	Description
FtpAddVirtualUser	Add a new virtual user for the specified server
FtpAuthenticateClient	Authenticate the client and assign access rights for the session
FtpChangeClientDirectory	Change the current working directory for the client session
FtpCreateServerCredentials	Create a new server security credentials structure
FtpDeleteServerCredentials	Delete a previously created security credentials structure
FtpDeleteVirtualUser	Delete a virtual user from the specified server
FtpDisconnectClient	Disconnect the specific client session, closing the control channel and aborting any file transfer
FtpEnableClientAccess	Enable or disable access rights for the specified client session
FtpEnableCommand	Enable or disable a specific server command
FtpEnumServerClients	Returns a list of active client sessions established with the specified server
FtpGetActiveClient	Return the client ID for the active client session associated with the current thread
FtpGetClientAccess	Return the access rights that have been granted to the client session
FtpGetClientAddress	Return the IP address of the specified client session
FtpGetClientCredentials	Return the credentials for the specified client session
FtpGetClientDirectory	Return the current working directory for a client session
FtpGetClientFileType	Return the current file type used for transfers by the specified client
FtpGetClientHomeDirectory	Return the home directory for an authenticated client session
FtpGetClientIdentity	Return the identity of the specified client session
FtpGetClientIdleTime	Return the idle timeout period for the specified client
FtpGetClientLocalPath	Return the full local path for the specified virtual path
FtpGetClientServer	Return the handle to the server that created the specified client session
FtpGetClientThreadId	Returns the thread ID associated with the specified client session
FtpGetClientUserName	Return the user name associated with the specified client session
FtpGetClientVirtualPath	Return the virtual path for a local file on the server
FtpGetCommandFile	Return the full path to the local file name or directory specified by the client
FtpGetCommandLine	Return the complete command line issued by the client
FtpGetCommandName	Return the name of the last command issued by the client
FtpGetCommandParam	Return the value of the specified parameter for the command issued by the client
FtpGetCommandParamCount	Return the number of parameters to the current command issued by the

	client
FtpGetCommandResult	Return the result code and a description of the last command processed by the server
FtpGetCommandUsage	Return the number of times a specific command has been issued by all clients
FtpGetProgramExitCode	Return the exit code of the last program executed by the client
FtpGetProgramName	Return the name of the last program executed by the client
FtpGetProgramOutput	Return a copy of the standard output from the last program executed by the client
FtpGetProgramText	Return a copy of the standard output from the last program in a string buffer
FtpGetRenamedFile	Return the original name of a file being renamed by the client
FtpGetServerAddress	Return the IP address for the server
FtpGetServerDirectory	Return the full path to the root directory assigned to the specified server
FtpGetServerError	Return information about the last server error that occurred
FtpGetServerIdentity	Return the identity and version information for the specified server
FtpGetServerLogFile	Return the current log file format and full path for the file
FtpGetServerMemoryUsage	Return the amount of memory allocated for the server and all client sessions
FtpGetServerName	Return the host name assigned to the server or specified client session
FtpGetServerOptions	Return the configuration options for the specified server
FtpGetServerPriority	Return the current priority assigned to the specified server
FtpGetServerStackSize	Return the initial size of the stack allocated for threads created by the server
FtpGetServerTransferInfo	Return information about the current file transfer for the client session
FtpGetServerUuid	Return the UUID assigned to the specified server
FtpGetServerUuidString	Return the UUID assigned to the server as a printable string
FtpIsClientAnonymous	Determine if the specified client has authenticated as an anonymous user
FtpIsClientAuthenticated	Determine if the specified client session has been authenticated
FtpIsCommandEnabled	Determine if the specified command is currently enabled or disabled
FtpRegisterProgram	Register a program for use with the SITE EXEC command
FtpRenameServerLogFile	Rename or delete the current log file being updated by the server
FtpSendResponse	Send a result code and optional message to the client in response to a command
FtpServerAsyncNotify	Enable or disable asynchronous notification of changes in server status
FtpServerDisableTrace	Disable logging of network function calls
FtpServerEnableTrace	Enable logging of network function calls to a file

FtpServerInitialize	Initialize the library and validate the specified license key at runtime
FtpServerProc	Callback function used to process server events
FtpServerRestart	Restart the server, terminating all active client sessions
FtpServerResume	Resume accepting client connections on the specified server
FtpServerStart	Start the server and begin accepting client connections
FtpServerStop	Stop the server and terminate all active client connections
FtpServerSuspend	Suspend accepting client connections on the specified server
FtpServerThrottle	Limit the number of active client connections, connections per address and connection rate
FtpServerUninitialize	Terminate use of the library by the application
FtpSetClientAccess	Change the access rights associated with the specified client session
FtpSetClientFileType	Change the current file type used for transfers by the specified client
FtpSetClientIdentity	Change the identity string associated with the specified client session
FtpSetClientIdleTime	Change the idle timeout period for the specified client session
FtpSetCommandFile	Change the name of the local file or directory that is the target of the current command
FtpSetServerAddress	Change the IP address that the server will use with passive data connections
FtpSetServerError	Set the last error code for the specified server session
FtpSetServerIdentity	Change the identity and version information for the specified server
FtpSetServerLogFile	Change the current log format, level of detail and file name
FtpSetServerName	Change the hostname assigned to the specified server or client session
FtpSetServerPriority	Change the priority assigned to the specified server
FtpSetServerStackSize	Change the initial size of the stack allocated for threads created by the server

FtpAddVirtualUser Function

```
BOOL WINAPI FtpAddVirtualUser(  
    HSERVER hServer,  
    UINT nHostId,  
    DWORD dwUserAccess,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszDirectory  
);
```

Add a new virtual user for the specified server.

Parameters

hServer

The server handle.

nHostId

An integer value which identifies the virtual host. This parameter is reserved for future use and must always have a value of zero.

dwUserAccess

An integer value which specifies the access clients will be given when authenticated as this user. For a list of user access permissions, see [User Access Constants](#).

lpszUserName

A pointer to a string which specifies the user name. The maximum length of a username is 63 characters and it is recommended that names be limited to alphanumeric characters. Whitespace, control characters and certain symbols such as path delimiters and wildcard characters are not permitted. If an invalid character is included in the name, the function will fail with an error indicating the username is invalid. This parameter cannot be NULL and the name must be at least three characters in length. Usernames are not case sensitive.

lpszPassword

A pointer to a string which specifies the user password. The maximum length of a password is 63 characters and is limited to printable characters. Whitespace and control characters are not permitted. If an invalid character is included in the password, the function will fail with an error indicating the password is invalid. This parameter cannot be NULL and must be at least one character in length. Passwords are case sensitive.

lpszDirectory

A pointer to a string which specifies the directory that will be the virtual user's home directory. If the server was started in multi-user mode, this directory will be relative to the user directory created by the server, otherwise it will be relative to the server root directory. If the directory does not exist, it will be created the first time that the virtual user successfully logs in to the server. If this parameter is NULL or an empty string, a default home directory will be created for the virtual user.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid or the virtual host ID does not specify a valid host, the function will return zero. If the function fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **FtpAddVirtualUser** function adds a virtual user that is associated with the specified virtual host. When a client connects with the server and provides authentication credentials, the server will check if the username has been created using this function. If a match is found, the client access rights will be updated.

If you wish to modify the information for a user, it is not necessary to delete the username first. If this function is called with a username that already exists, that record is replaced with the values passed to this function. You cannot use this function to create a virtual user named "anonymous".

The virtual users created by this function exist only as long as the server is active. If you wish to maintain a persistent database of users and passwords, you are responsible for its implementation based on the requirements of your specific application. For example, a simple implementation would be to store the user information in a local XML or INI file and then read that configuration file after the server has started, calling this function for each user that is listed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpDeleteVirtualUser](#)

FtpAuthenticateClient Function

```
BOOL WINAPI FtpAuthenticateClient(  
    HSERVER hServer,  
    UINT nClientId,  
    DWORD dwUserAccess,  
    BOOL bCreateHome,  
    LPCTSTR lpszDirectory  
);
```

Authenticate the client and assign access rights for the session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies one or more user access rights. For a list of user access rights that can be granted to the client, see [User Access Constants](#).

bCreateHome

An integer value that specifies if the server should create the home directory for the authenticated client if it does not already exist. If this value is non-zero, the home directory will be created. If value is zero, the home directory will not be created and if it does not exist, this function will fail.

lpszDirectory

A pointer to a string that specifies the home directory for the user. If an absolute path is specified, it will be relative to the server root directory. If a relative path is specified, it will be a subdirectory of the home directory for the server instance. If this parameter is NULL or an empty string, a home directory will be assigned based on the server home directory and the user name.

Return Value

If the the client session could be authenticated, the return value is non-zero. If the server handle and client ID do not specify a valid client session, or the client has already been authenticated, this function will return zero.

Remarks

The **FtpAuthenticateClient** function is used to authenticate a specific client session, typically in response to an FTP_CLIENT_USERAUTH event that indicates a client has requested authentication. This function is also used internally to automatically grant the appropriate access rights to local user and anonymous client sessions.

It is recommended that most applications specify FTP_ACCESS_DEFAULT as the *dwUserAccess* value for a client session, since this allows the server automatically grant the appropriate access based on the server configuration options for normal and anonymous users. If the server is going to be publicly accessible or third-party FTP clients will be used to access the server, you should always grant the FTP_ACCESS_LIST permission to clients. Many client applications will not function correctly if they are unable to obtain a list of files in the user's home directory.

If `FTP_ACCESS_RESTRICTED` is specified and the server was started in multi-user mode, the client session will be effectively locked to its home directory and cannot navigate to the server root directory. By default, restricted client sessions are also limited to only downloading files and requesting directory listings. If a client session is not restricted, the client can access files outside of its home directory. Regardless of this option, a client cannot access files outside of the server root directory.

If `FTP_ACCESS_RESTRICTED` or `FTP_ACCESS_ANONYMOUS` is specified, the client session will be authenticated in a restricted mode and the access rights for the session will persist until the client disconnects from the server. Unlike regular users, the access rights for a restricted client cannot be changed by the server at a later point. This restriction is designed to prevent the inadvertent granting of rights to an untrusted client that could compromise the security of the server.

If the *lpszDirectory* parameter is `NULL` or an empty string and the server has been started in multi-user mode, each user is assigned their own home directory based on their username. If the server has not been started in multi-user mode, then the default home directory will be the server root directory and is shared by all users. The **FtpGetClientHomeDirectory** function will return the full path to the home directory for an authenticated client.

If the `FTP_ACCESS_EXECUTE` permission is granted to the client session, it can execute external programs using the `SITE EXEC` command. Because the program is executed in the context of the server process, it is recommended that you limit access to this functionality and ensure that the programs being executed do not introduce any security risks to the operating system. This permission is never granted by default, and the `SITE EXEC` command will return an error if the client session is anonymous, regardless of whether this permission is granted or not.

This function should only be used for custom authentication schemes and is not necessary if you have used the **FtpAddVirtualUser** function to create virtual users.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpAddVirtualUser](#), [FtpChangeClientDirectory](#), [FtpGetClientCredentials](#), [FtpGetClientDirectory](#), [FtpGetClientHomeDirectory](#)

FtpChangeClientDirectory Function

```
BOOL WINAPI FtpChangeClientDirectory(  
    HSERVER hServer,  
    UINT nClientId,  
    LPCTSTR lpszDirectory  
);
```

Change the current working directory for the specified client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszDirectory

A pointer to a string which specifies the new current working directory for the client session. If this parameter is NULL or an empty string, the current working directory will be changed to the client home directory. If this parameter is not NULL, it must specify a directory that exists and is accessible by the server process.

Return Value

If the current working directory was changed, the return value is non-zero. If the server handle and client ID do not specify a valid client session, or the directory is invalid, this function will return zero.

Remarks

The **FtpChangeClientDirectory** function will change the current working directory for the specified client session. This function is called internally when the client sends the CWD or CDUP commands, however it may be explicitly used by the application to change the client's working directory in response to a server event.

This function cannot be used to change the current working directory for a client to an arbitrary directory outside of the server root directory. If the *lpszDirectory* parameter specifies a relative path (i.e.: a path that does not begin with a drive letter or leading path delimiter) then the new working directory will be relative to the current working directory. If an absolute path is specified, the absolute path must include the complete path to either the server root directory or the user's home directory, based on the permissions granted to the client session. If a path outside of the server root directory is specified, this function will fail with an access denied error.

Use caution when calling this function to override the directory specified by the client when it sends the CWD or CDUP commands. If your application changes the current working directory to one not specified by the client, it may cause unpredictable behavior in the client application because the actual path of the current working directory will not match the directory that was requested.

If this function is used to change the current working directory in response to the CWD command, you should not call the **FtpGetCommandParam** function and pass the command parameter as an argument to this function. You must use the **FtpGetCommandFile** function to obtain the directory name provided by the client prior to calling this function.

The application should never call the **SetCurrentDirectory** function in the Windows API to change

the current directory for the process to the working directory of a client session. Because the server is multithreaded and each client session is managed in its own thread, an application using this library should avoid using relative paths.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetClientDirectory](#), [FtpGetClientHomeDirectory](#), [FtpGetCommandFile](#)

FtpCreateServerCredentials Function

```
BOOL WINAPI FtpCreateServerCredentials(  
    DWORD dwProtocol,  
    DWORD dwOptions,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName,  
    LPVOID lpvReserved,  
    LPSECURITYCREDENTIALS* lppCredentials  
);
```

The **FtpCreateServerCredentials** function creates a SECURITYCREDENTIALS structure.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is

	supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A pointer to a string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A pointer to a string which specifies the certificate owner's password. A value of NULL specifies

that no password is required. This parameter is only used if a PKCS12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpzCertStore

A pointer to a string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.

lpzCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

lpvReserved

Pointer reserved for future use. Set it to NULL when using this function.

lppCredentials

Pointer to an [LPSECURITYCREDENTIALS](#) pointer. The memory for the credentials structure will be allocated by this function and must be released by calling the **FtpDeleteServerCredentials** function when it is no longer needed. The pointer value must be set to NULL before the function is called. It is important to note that this is a pointer to a pointer variable, not a pointer to the SECURITYCREDENTIALS structure itself.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetServerError**.

Remarks

The structure that is created by this function may be used as client credentials when establishing a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can

be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **FtpServerStart** function.

Example

```
// Create the server credentials that identifies the certificate
// that will be used for secure connections
LPSECURITYCREDENTIALS lpSecCred = NULL;

FtpCreateServerCredentials(SEcurity_PROTOCOL_DEFAULT,
                           0,
                           NULL,
                           NULL,
                           lpzCertStore,
                           lpzCertName,
                           NULL,
                           &lpSecCred);

// Start the server
hServer = FtpServerStart(lpzLocalHost,
                        FTP_PORT_DEFAULT,
                        FTP_SERVER_LOCALUSER | FTP_SERVER_UNIXMODE |
FTP_SERVER_SECURE,
                        &ftpConfig,
                        lpEventHandler,
                        0,
                        lpSecCred);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpDeleteServerCredentials](#), [FtpServerStart](#), [SECURITYCREDENTIALS](#)

FtpDeleteServerCredentials Function

```
VOID WINAPI FtpDeleteServerCredentials(  
    LPSECURITYCREDENTIALS* LppCredentials  
);
```

The **FtpDeleteServerCredentials** function deletes an existing **SECURITYCREDENTIALS** structure.

Parameters

lppCredentials

Pointer to an LPSECURITYCREDENTIALS pointer. On exit from the function, the pointer value will be NULL.

Return Value

None.

Example

```
if (lpSecCred != NULL)  
    FtpDeleteServerCredentials(&lpSecCred);
```

Remarks

This function can be used to release the memory allocated for the credentials after the server has been started.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpCreateServerCredentials](#), [FtpServerStop](#)

FtpDeleteVirtualUser Function

```
BOOL WINAPI FtpDeleteVirtualUser(  
    HSERVER hServer,  
    UINT nHostId,  
    LPCTSTR LpszUserName  
);
```

Remove a virtual user from the specified host.

Parameters

hServer

The server handle.

nHostId

An integer value which identifies the virtual host. This parameter is reserved for future use and must always have a value of zero.

lpszUserName

A pointer to a string which specifies the user that will be removed. This parameter cannot be a NULL pointer or an empty string.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid or the virtual host ID does not specify a valid host, the function will return zero. If the function fails, the last error code will be updated to indicate the cause of the failure.

Remarks

This function removes a virtual user that was created by a previous call to the **FtpAddVirtualUser** function. This function will not match partial usernames and wildcard characters cannot be used to delete multiple users. Usernames are not case sensitive. You cannot use this function to delete the "anonymous" user.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpAddVirtualUser](#)

FtpDisconnectClient Function

```
BOOL WINAPI FtpDisconnectClient(  
    HSERVER hServer,  
    UINT nClientId  
);
```

Close the control connection for the specified client and release the resources allocated for the session

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **FtpDisconnectClient** function will close the control channel, disconnecting the client from the server and terminating the client session thread. Resources that we allocated for the client, such as memory and open handles, will be released back to the operating system. If the client was in the process of transferring a file, the transfer will be aborted. This performs the same operation as if the client sent the QUIT command to the server.

This function sends an internal control message that notifies the server that this session should be terminated. When the session thread is signaled that it should terminate, it will abort any active file transfers and begin to release the resources allocated for that session. To ensure that the client session terminates gracefully, there may be a brief period of time where the session thread is still active after this function has returned.

After this function returns, the application should never use the same client ID with another function. Client IDs are unique to the session over the lifetime of the server, and are not reused.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

See Also

[FtpServerRestart](#), [FtpServerStop](#)

FtpEnableClientAccess Function

```
BOOL WINAPI FtpEnableClientAccess(  
    HSERVER hServer,  
    UINT nClientId,  
    DWORD dwUserAccess,  
    BOOL bEnable  
);
```

Enable or disable access rights for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies an access right to enable or disable. For a list of user access rights that can be granted to the client, see [User Access Constants](#).

bEnable

An integer value which specifies if permission should be granted or revoked for the specified access right. If this value is non-zero, permission is granted to the client to perform the action specified by the *dwUserAccess* parameter. If this value is zero, that permission is revoked.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero. This function can only be used with authenticated clients. If the client session has not been authenticated, the return value will be zero.

Remarks

The **FtpEnableClientAccess** function is used to enable or disable access to specific functionality by the client. The function can only change a single access right and cannot be used to enable or disable multiple access rights in a single function call. To change multiple user access rights for the client, use the **FtpSetClientAccess** function.

This function cannot be used to change the access rights for a restricted or anonymous user. Those rights are granted when the client session is authenticated and will persist until the client disconnects from the server. This restriction is designed to prevent the inadvertent granting of rights to an untrusted client that could compromise the security of the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

See Also

[FtpAuthenticateClient](#), [FtpGetClientAccess](#), [FtpSetClientAccess](#)

FtpEnableCommand Function

```
BOOL WINAPI FtpEnableCommand(  
    HSERVER hServer,  
    LPCTSTR lpszCommand,  
    BOOL bEnable  
);
```

Enable or disable a specific server command

Parameters

hServer

The server handle.

lpszCommand

A pointer to a NULL terminated string that specifies the name of the command to be enabled or disabled. The command name is not case-sensitive, but the value must otherwise match the exact name. Partial matches are not recognized by this function. This parameter cannot be NULL.

bEnable

An integer value which specifies if the command should be enabled or disabled. If the value is non-zero, the command is enabled. If the value is zero, the command will be disabled.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid or the command is not recognized, the function will return zero. If the function fails, the **FtpGetServerError** function will return more information about the last error that has occurred.

Remarks

The **FtpEnableCommand** function is used to enable or disable access to a specific command on the server. When a command is disabled, it will also disable any corresponding feature related to that command. For example, if the MDTM command is disabled and a client issues the FEAT command to request a list of supported features, the command will no longer be listed. This function is typically used to enable or disable certain commands for compatibility with older client software. The **FtpIsCommandEnabled** function can be used to determine if a command is enabled or not.

The command name provided to this function must match the commands defined in RFC 959 or related protocol standards. It is important to distinguish between commands recognized by an FTP server and the commands that client programs may use. For example, the standard Windows FTP command line program provides commands such as GET and PUT to download and upload files. However, those are not the actual commands sent to a server. Instead, the corresponding server commands issued by a client application would be RETR (retrieve) and STOR (store). Refer to [File Transfer Protocol Commands](#) for a complete list of server commands.

Some commands cannot be disabled because they are required to perform essential server functions. For example, the USER and PASS commands are required to perform client authentication and therefore cannot be disabled. If you attempt to disable a required command, this function will return zero and the last error code will be set to ST_ERROR_COMMAND_REQUIRED. Because this function affects all clients connected to the server, it should not be used to limit access to certain commands for specific clients. Instead, either assign the client the appropriate permissions using the **FtpAuthenticateClient** function, or use an

event handler to filter the commands.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpAuthenticateClient](#), [FtpIsCommandEnabled](#)

FtpEnumServerClients Function

```
INT WINAPI FtpEnumServerClients(  
    HSERVER hServer,  
    DWORD dwReserved  
    UINT * lpClients,  
    INT nMaxClients  
);
```

Return a list of active client sessions established with the specified server.

Parameters

hServer

Handle to the server socket.

dwReserved

An unsigned integer value that is reserved for future use. This parameter should always be zero.

lpClients

Pointer to an array of unsigned integers which will contain client IDs that uniquely identifies each client when the function returns. If this parameter is NULL, then the function will return the number of active client connections established with the server.

nMaxClients

Maximum number of client IDs to be returned in the *lpClients* array. If the *lpClients* parameter is NULL, this parameter should have a value of zero.

Return Value

If the function succeeds, the return value is the number of active client connections to the server. If the function fails, the return value is FTP_ERROR. To get extended error information, call

FtpGetServerError.

Remarks

If the *nMaxClients* parameter is less than the number of active client connections, the function will fail and the last error code will be set to the error ST_ERROR_BUFFER_TOO_SMALL. To dynamically determine the number of active connections, call the function with the *lpClients* parameter with a value of NULL, and the *nMaxClients* parameter with a value of zero.

Example

```
INT nClients = FtpEnumServerClients(hServer, 0, NULL, 0);  
  
if (nClients > 0)  
{  
    UINT *lpClients = NULL;  
  
    // Allocate memory for an array of client IDs  
    lpClients = (UINT *)LocalAlloc(LPTR, nClients * sizeof(UINT));  
  
    if (lpClients == NULL)  
    {  
        // Virtual memory has been exhausted  
        return;  
    }  
  
    nClients = FtpEnumServerClients(hServer, 0, lpClients, nClients);  
}
```

```
if (nClients == FTP_ERROR)
{
    // Unable to obtain list of connected clients
    return;
}

for (INT nIndex = 0; nIndex < nClients; nIndex++)
{
    TCHAR szUserName[FTP_MAXUSERNAME];

    if (FtpGetClientUserName(hServer, lpClients[nIndex], szUserName,
FTP_MAXUSERNAME))
    {
        // Perform some action with the client user name
    }
}

// Free the memory allocated for the client IDs
LocalFree((HLOCAL)lpClients);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

See Also

[FtpServerStart](#)

FtpGetActiveClient Function

```
UINT WINAPI FtpGetActiveClient(  
    HSERVER hServer  
);
```

Return the client ID for the active client session associated with the current thread.

Parameters

hServer

The server handle.

Return Value

If the function succeeds, the return value is the unique ID associated with the client session for the current thread. If the server handle is invalid or there is no client session active on the current thread, the return value is zero.

Remarks

The **FtpGetActiveClient** function is used to obtain the client ID associated with the current thread. This means this function will only return a client ID if it is called within an event handler or a function called by an event handler. If this function is called by a function that is not executing within the context of an event handler it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[FtpEnumServerClients](#), [FtpGetClientThreadId](#)

FtpGetClientAccess Function

```
BOOL WINAPI FtpGetClientAccess(  
    HSERVER hServer,  
    UINT nClientId,  
    LPDWORD lpdwUserAccess  
);
```

Return the access rights that have been granted to the client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpdwUserAccess

A pointer to an unsigned integer which specifies one or more access rights for the client session. For a list of user access rights that can be granted to the client, see [User Access Constants](#). This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero. This function can only be used with authenticated clients. If the client session has not been authenticated, the return value will be zero.

Remarks

The **FtpGetClientAccess** function is used to obtain all of the access rights that are currently granted to an authenticated client session. The **FtpEnableClientAccess** function can be used to enable or disable specific permissions, and the **FtpSetClientAccess** function can change multiple access rights at once.

Example

```
DWORD dwUserAccess = 0;  
  
// Check if the client is a restricted user  
  
if (FtpGetClientAccess(hServer, nClientId, &dwUserAccess))  
{  
    if (dwUserAccess & FTP_ACCESS_RESTRICTED)  
    {  
        _tprintf(_T("Client %u authenticated as a restricted user\n"),  
nClientId);  
        return;  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

See Also

[FtpAuthenticateClient](#), [FtpEnableClientAccess](#), [FtpSetClientAccess](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

FtpGetClientAddress Function

```
INT WINAPI FtpGetClientAddress(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

Return the IP address of the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszAddress

A pointer to a string buffer that will contain the client IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the server handle or the client ID is invalid, or the buffer is not large enough to store the complete address, the function will return a value of zero.

Remarks

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetServerAddress](#)

FtpGetClientCredentials Function

```
BOOL WINAPI FtpGetClientCredentials(  
    HSERVER hServer,  
    UINT nClientId,  
    LPFTPCLIENTCREDENTIALS lpCredentials  
);
```

Return the user credentials for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpCredentials

A pointer to an **FTPCLIENTCREDENTIALS** structure that will contain information about the user when the function returns. This parameter cannot be NULL.

Return Value

If the user credentials for the client session are available, the return value is non-zero. If the server handle and client ID do not specify a valid client session, or the client has not requested authentication, this function will return zero.

Remarks

The **FtpGetClientCredentials** function is used to obtain the username and password that was provided by the client when it requested authentication. Typically this function is used in an event handler to validate the credentials provided by the client. If the credentials are considered valid, the event handler would then call the **FtpAuthenticateClient** function to specify that the session has been authenticated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpAuthenticateClient](#), [FTPCLIENTCREDENTIALS](#)

FtpGetClientDirectory Function

```
INT WINAPI FtpGetClientDirectory(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR LpszDirectory,  
    INT nMaxLength  
);
```

Returns the current working directory for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszDirectory

A pointer to a string buffer that will contain the current working directory for the specified client session, terminated with a null character. This buffer should be at least MAX_PATH characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be larger than zero or the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

This function returns the full path to the current working directory for the specified client session. For example, if the server root directory is C:\ProgramData\MyServer and the current working directory for the client is /Research/Documents, this function will return C:\ProgramData\MyServer\Research\Documents as the current working directory for the client session.

It is important to note that the current working directory for client sessions is virtual, and does not reflect the current working directory for the server process. To change the current working directory for a client, use the **FtpChangeClientDirectory** function.

This function should only be used with client sessions that have been authenticated. Unauthenticated clients are not assigned a current working directory and this function will return zero, with the last error code set to ST_ERROR_AUTHENTICATION_REQUIRED.

To convert a full path to the virtual path for a specific client session, use the **FtpGetClientVirtualPath** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpChangeClientDirectory](#), [FtpGetClientHomeDirectory](#), [FtpGetClientVirtualPath](#)

FtpGetClientFileType Function

```
BOOL WINAPI FtpGetClientFileType(  
    HSERVER hServer,  
    UINT nClientId,  
    UINT * lpnFileType  
);
```

Return the current file type used for transfers by the specified client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpnFileType

A pointer to an unsigned integer value that will contain the current file type used by the client for data transfers. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **FtpGetClientFileType** function will return the current file type that has been specified by the client sending the TYPE command to the server. The file type determines if there is any conversion performed on the data that is being exchanged between the client and server. The following file types are supported:

Value	Description
FILE_TYPE_ASCII (1)	The file is a text file using the ASCII character set. For those clients which use a different end-of-line character sequence, the text file has been converted to the local format which uses the carriage return (CR) and linefeed (LF) characters.
FILE_TYPE_IMAGE (3)	The file is a binary file and no data conversion of any type has been performed on the file. This is the default file type for most data files and executable programs. If the client specified this file type when appending to a text file, the file will contain the end-of-line sequences used by its native operating system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

See Also

[FtpSetClientFileType](#)

FtpGetClientHomeDirectory Function

```
INT WINAPI FtpGetClientHomeDirectory(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR lpszDirectory,  
    INT nMaxLength  
);
```

Returns the home directory for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszDirectory

A pointer to a string buffer that will contain the home directory for the specified client session, terminated with a null character. This buffer should be at least MAX_PATH characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be larger than zero or the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

This function returns the full path to the home working directory assigned to the specified client session. This will be the same path to the home directory specified when the **FtpAuthenticateClient** function was used to authenticate the client session. If a home directory was not explicitly assigned when the client was authenticated, then this function returns the default home directory that was created for the client, or the server root directory if the FTP_SERVER_MULTUSER option was not specified when the server was started.

This function should only be used with client sessions that have been authenticated. Unauthenticated clients are not assigned a home directory and this function will return zero, with the last error code set to ST_ERROR_AUTHENTICATION_REQUIRED.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

FtpGetClientIdentity Function

```
INT WINAPI FtpGetClientIdentity(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR lpszIdentity,  
    INT nMaxLength  
);
```

Return the identity of the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszIdentity

A pointer to a string buffer that will contain the identity of the client when the function returns, terminated with a null character. This parameter cannot be NULL. It is recommended that this buffer be at least 32 characters in length.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle is invalid, or the buffer is not large enough to store the complete path, the function will return a value of zero. If the client did not identify itself, this function will return zero.

Remarks

The **FtpGetClientIdentity** function returns the string that the client used to identify itself to the server. The client may use either the CLNT or CSID command to identify itself. Although the CLNT command is considered to be deprecated, it is supported for backwards compatibility with older clients. The identity string does not have any standard format and is used for informational purposes only and does not affect the operation of the server in any way. Not all clients identify themselves, in which case this function will return zero and the *lpszIdentity* string buffer will be set to an empty string.

If the client does identify itself, it typically uses the name of the client application that was used to establish the connection. The application may choose to assign an identity to a client session for its own internal purposes using the **FtpSetClientIdentity** function, regardless of whether the client identifies itself.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetServerIdentity](#), [FtpSetClientIdentity](#), [FtpSetServerIdentity](#)

FtpGetClientIdleTime Function

```
UINT WINAPI FtpGetClientIdleTime(  
    HSERVER hServer,  
    UINT nClientId,  
    UINT * lpnElapsed  
);
```

Return the idle timeout period for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpnElapsed

A pointer to an unsigned integer value that will contain the number of seconds the client session has been idle. This parameter may be NULL if this information is not required.

Return Value

If the function succeeds, the return value is client idle timeout period in seconds. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **FtpGetClientIdleTime** function will return the number of seconds that the client may remain idle before being automatically disconnected by the server. The idle time of a client session is based on the last time a command was issued to the server or when a file data transfer completed. The server will never disconnect a client that is in the process of uploading or downloading a file, regardless of the idle timeout period.

The default idle timeout period for a client is 900 seconds (15 minutes), however the server can be configured to use a different value and individual clients can request the timeout period be changed by sending the IDLE command to the server. For a client to be able to change its own timeout period, it must be granted the FTP_ACCESS_IDLE permission. The minimum timeout period for a client is 60 seconds, the maximum is 7200 seconds (2 hours). An application can change the timeout period for a specific client session using the **FtpSetClientIdleTime** function.

It is important to note that the idle timeout period only affects authenticated clients.

Unauthenticated clients use a different internal timer that limits the amount of time they can remain connected to the server before successfully authenticating with a valid username and password. By default, the authentication timeout period is 60 seconds and is set when the server is started; it cannot be changed for an individual client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

See Also

[FtpSetClientIdleTime](#)

FtpGetClientLocalPath Function

```
INT WINAPI FtpGetClientLocalPath(  
    HSERVER hServer,  
    UINT nClientId,  
    LPCTSTR lpszVirtualPath,  
    LPTSTR lpszLocalPath,  
    INT nMaxLength,  
);
```

Return the full local path for a virtual filename or directory on the server.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszVirtualPath

A pointer to a string that specifies an virtual path on the server. This parameter cannot be NULL.

lpszLocalPath

A pointer to a string buffer that will contain the full local path, terminated with a null-character. This buffer should be at least MAX_PATH characters to accommodate the complete path. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero. If the string buffer is not large enough to contain the complete path, this function will return zero and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL.

Remarks

The **FtpGetClientLocalPath** function takes a virtual path and returns the full path to the specified file or directory on the local system. The virtual path may be absolute or relative to the current working directory for the client session. This function will recognize a tilde at the beginning of the path to specify the client home directory.

To obtain the virtual path for a local file or directory, use the **FtpGetClientVirtualPath** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

FtpGetClientServer Function

```
HSERVER WINAPI FtpGetClientServer(  
    UINT nClientId  
);
```

The **FtpGetClientServer** function returns a handle to the server that created the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the function succeeds, the return value is the handle to the server that created the client session.

If the function fails, the return value is `INVALID_SERVER`. To get extended error information, call the **FtpGetServerError** function.

Remarks

The **FtpGetClientServer** function returns the handle to the server that created the client session and is typically used within a notification message handler. If the server is in the process of shutting down, or the client session thread is terminating, this function will fail and return `INVALID_SERVER` indicating that the session ID is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `csftools10.h`.

Import Library: `csftsv10.lib`

See Also

[FtpServerAsyncNotify](#)

FtpGetClientThreadId Function

```
DWORD WINAPI FtpGetClientThreadId(  
    HSERVER hServer,  
    UINT nClientId  
);
```

Returns the thread ID associated with the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the function succeeds, the return value is a thread ID. If the function fails, the return value is zero. To get extended error information, call the **FtpGetServerError** function.

Remarks

The **FtpGetClientThreadId** function returns a thread ID that can be used to identify the thread that is managing the client session. The thread ID can be used with other Windows API functions such as **OpenThread**. Exercise caution when using thread-related functions, interfering with the normal operation of the thread can have unexpected results. You should never use this function to obtain a thread handle and then call the **TerminateThread** function to terminate a client session. This will prevent the thread from releasing the resources that were allocated for the session and can leave the server in an unstable state. To terminate a client session, use the **FtpDisconnectClient** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

See Also

[FtpEnumServerClients](#), [FtpGetActiveClient](#)

FtpGetClientUserName Function

```
INT WINAPI FtpGetClientUserName(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR LpszUserName,  
    INT nMaxLength  
);
```

Return the user name associated with the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszUserName

A pointer to a string buffer that will contain the user name associated with the client session. This buffer must be large enough to store the complete user name, including the terminating null character. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This parameter must have a value larger than zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **FtpGetClientUserName** function returns the user name associated with an authenticated client session. If the client has not authenticated itself by sending the USER and PASS commands, this function will return zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpAuthenticateClient](#), [FtpGetClientAccess](#), [FtpGetClientHomeDirectory](#)

FtpGetClientVirtualPath Function

```
INT WINAPI FtpGetClientVirtualPath(  
    HSERVER hServer,  
    UINT nClientId,  
    LPCTSTR lpszLocalPath,  
    LPTSTR lpszVirtualPath,  
    INT nMaxLength,  
);
```

Return the virtual path for a local file on the server.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszLocalPath

A pointer to a string that specifies an absolute path on the local system. This parameter cannot be NULL.

lpszVirtualPath

A pointer to a string buffer that will contain the virtual path, terminated with a null-character. This buffer should be at least MAX_PATH characters to accommodate the complete path. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero. If the string buffer is not large enough to contain the complete path, this function will return zero and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL.

Remarks

A virtual path for the client is either relative to the server root directory, or the client home directory if the client was authenticated as a restricted user. These virtual paths are what the client will see as an absolute path on the server. For example, if the server was configured to use "C:\ProgramData\MyServer" as the root directory, and the *lpszLocalPath* parameter was specified as "C:\ProgramData\MyServer\Documents\Research", this function would return the virtual path to that directory as "/Documents/Research".

If the client session was authenticated as a restricted user, then the virtual path is always relative to the client home directory instead of the server root directory. This is because restricted users are isolated to their own home directory and any subdirectories. For example, if restricted user "John" has a home directory of "C:\ProgramData\MyServer\Users\John" and the *lpszLocalPath* parameter was specified as "C:\ProgramData\MyServer\Users\John\Accounting\Projections.pdf" this function would return the virtual path as "/Accounting/Projections.pdf".

If the *lpszLocalPath* parameter specifies a file or directory outside of the server root directory, this function will return zero and the last error code will be set to ST_ERROR_INVALID_FILE_NAME. This function can only be used with authenticated clients. If the *nClientId* parameter specifies a client session that has not been authenticated, this function will return zero and the last error code will be ST_ERROR_AUTHENTICATION_REQUIRED.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetCommandFile](#), [FtpGetClientLocalPath](#)

FtpGetCommandFile Function

```
INT WINAPI FtpGetCommandFile(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR lpszFileName,  
    INT nMaxLength  
);
```

Get the full path to the local file name or directory specified by the client

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A pointer to a string buffer that will contain the full path to a file name or directory specified by the client when it issued a command. The string buffer will be null terminated and must be large enough to store the complete file path. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. It is recommended that the buffer be at least MAX_PATH characters in size. If the maximum length specified is smaller than the actual length of the full path, this function will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the function fails, the return value will be zero and the

FtpGetServerError function can be used to retrieve the last error code. If the last error code is returned as a value of zero, this means that the command issued by the client accepts a file name as an argument, but the client did not specify one.

Remarks

The **FtpGetCommandFile** function is used to obtain the full path to a local file name or directory specified by the client as an argument to a standard FTP command. For example, if the client sends the RETR command to the server, this function will return the complete path to the local file that the client wants to download. This function will only work with those standard commands that perform some action on a file or directory.

This function should always be used to obtain the file name for a command that performs a file or directory operation. The **FtpGetCommandParam** function will return the actual command parameter, but the file name will typically be relative to the user home directory or server root directory, and cannot be passed directly to a Windows API function. The **FtpGetCommandFile** function normalizes the path provided by the client and ensures that it specifies a file or directory name in the correct location.

To change the file or directory name that is the target of the current command, use the **FtpSetCommandFile** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetClientDirectory](#), [FtpGetClientHomeDirectory](#), [FtpGetCommandLine](#), [FtpGetCommandParam](#),
[FtpSetCommandFile](#)

FtpGetCommandLine Function

```
INT WINAPI FtpGetCommandLine(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR LpszCmdLine,  
    INT nMaxLength  
);
```

Return the complete command line issued by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCmdLine

A pointer to a string buffer that will contain the command, including all arguments. The string buffer will be null terminated and must be large enough to store the complete command line. If this parameter is NULL, the function will return the length of the command line.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. The internal limit on the maximum length of a command is 1024 characters. If the maximum length specified is smaller than the actual length of the complete command, this function will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the function fails, the return value will be zero and the

FtpGetServerError function can be used to retrieve the last error code. If the last error code has a value of zero then no command has been issued by the client.

Remarks

The **FtpGetCommandLine** function is used to obtain the command that was issued by the client, and is commonly used inside FTP_CLIENT_COMMAND and FTP_CLIENT_RESULT event handlers to pre-process and post-process client commands, respectively. When the function returns, the string buffer provided by the caller will contain the complete command, including all command parameters. Any extraneous whitespace will be removed, however quoted parameters will be retained as-is.

To obtain a specific parameter to a command, use the **FtpGetCommandParam** function. The **FtpGetCommandParamCount** function will return the number of command parameters that were provided by the client. If the command sent by the client is used to perform an action on a file or directory, the **FtpGetCommandFile** function should be called to obtain the full path to the specified file rather than using the value of the command parameter.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetCommandFile](#), [FtpGetCommandParam](#)

FtpGetCommandName Function

```
INT WINAPI FtpGetCommandName
    HSERVER hServer,
    UINT nClientId,
    LPTSTR lpszCommand,
    INT nMaxLength
);
```

Return the name of the last command issued by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCommand

A pointer to a string buffer that will contain the command name. The string buffer will be null terminated and must be large enough to store the complete command name. If this parameter is NULL, the function will only return the length of the current command in characters, not including the terminating null character.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. If the maximum length specified is smaller than the actual length of the parameter, this function will fail. If the *lpszCommand* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the function fails, the return value will be zero and the **FtpGetServerError** function can be used to retrieve the last error code. If the last error code is returned as a value of zero, this means that no command has been issued by the client.

Remarks

The **FtpGetCommandName** function is used to obtain the name of the last command that was issued by the client. The command name returned by this function will always be capitalized, regardless of how it was sent by the client. This function is typically used inside `FTP_CLIENT_COMMAND` and `FTP_CLIENT_RESULT` event handlers to pre-process and post-process client commands, respectively.

The **FtpGetCommandParam** function can be used to return the value of individual command parameters specified by the client. The **FtpGetComandLine** function can be used to obtain the complete command line issued by the client.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetCommandFile](#), [FtpGetCommandLine](#), [FtpGetCommandParam](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

FtpGetCommandParam Function

```
INT WINAPI FtpGetCommandParam
    HSERVER hServer,
    UINT nClientId,
    INT nParam,
    LPTSTR lpszParam,
    INT nMaxLength
);
```

Return the value of the specified command parameter from the last command issued by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

nParam

An integer value which specifies the command parameter. A value of zero specifies the command itself, while values greater than zero specify a particular parameter. This function will fail if this value is less than zero or greater than the number of parameters available.

lpszParam

A pointer to a string buffer that will contain the command parameter. The string buffer will be null terminated and must be large enough to store the complete parameter value. If this parameter is NULL, the function will only return the length of the specified parameter in characters, not including the terminating null character.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. The internal limit on the maximum length of a command is 1024 characters. If the maximum length specified is smaller than the actual length of the parameter, this function will fail. If the *lpszParam* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the function fails, the return value will be zero and the **FtpGetServerError** function can be used to retrieve the last error code. If the last error code is returned as a value of zero, this means that no command has been issued by the client.

Remarks

The **FtpGetCommandParam** function is used to obtain a specific parameter for the last command that was issued by the client. If the parameter was surrounded in quotes, those quotes will be included in the value returned by this function. This function is typically used inside FTP_CLIENT_COMMAND and FTP_CLIENT_RESULT event handlers to pre-process and post-process client commands, respectively.

The **FtpGetCommandParamCount** function will return the number of command parameters that were provided by the client. If the command sent by the client is used to perform an action on a file or directory, the **FtpGetCommandFile** function should be called to obtain the full path to the

specified file rather than using the value of the command parameter. To obtain the complete command line, use the **FtpGetCommandLine** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetCommandFile](#), [FtpGetCommandLine](#), [FtpGetCommandName](#),
[FtpGetCommandParamCount](#)

FtpGetCommandParamCount Function

```
INT WINAPI FtpGetCommandParamCount
    HSERVER hServer,
    UINT nClientId
);
```

Return the number of command parameters for the last command issued by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

An integer value which specifies the number of parameters that were specified in the last command issued by the client. If the command did not include any parameters, this function will return zero. If the client has not issued a command, or the client session ID is invalid, this function will return -1.

Remarks

The **FtpGetCommandParamCount** function is used to determine the number of parameters specified in the last client command, and the maximum value that may be passed as the parameter index to the **FtpGetCommandParam** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

See Also

[FtpGetCommandLine](#), [FtpGetCommandName](#), [FtpGetCommandParam](#)

FtpGetCommandResult Function

```
INT WINAPI FtpGetCommandResult(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR lpszResult,  
    INT nMaxLength  
);
```

Return the result code and description for the last command issued by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszResult

A pointer to a string buffer that will contain the description of the result code. The string buffer will be null terminated up to the maximum number of characters specified by the caller. This parameter can be NULL if this information is not required.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. If the *lpszResult* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the result code for the last command issued by the client. A return value of zero indicates that the command has not completed and there is no result code available.

Remarks

The **FtpGetCommandResult** function is used to determine the result of the last command that was issued by the client and is typically called in the `FTP_CLIENT_RESULT` event handler. This function should only be called after a command has been processed or the **FtpSendResponse** function has been called.

The result code is a three-digit integer value that indicates the success or failure of a command. Whenever a client sends a command to the server, the server must respond with this numeric code, and optionally a text message that further describes the result. The text message may be a single line, or it may span multiple lines, with each line of text terminated by a carriage return and linefeed. Result codes are generally broken down into the following categories:

Result Code	Description
100-199	Result codes in this range indicate that the requested action is being initiated, and the client should expect another reply from the server before proceeding. This is normally used with file transfers, indicating to the client that the data transfer has started.
200-299	Result codes in this range indicate that the server has successfully completed the requested action. One exception is the 202 result code which indicates that the command is not implemented, but the client should not consider this to be an error condition.

300-399	Result codes in this range indicate that the requested action cannot complete until additional information is provided to the server. This is normally used with commands that require a specific sequence to complete. For example, the server will send the 331 result code in response to the USER command, which tells the client that it must send the PASS command to complete the authentication process.
400-499	Result codes in this range indicate that the requested action did not take place, but the error condition is temporary and may be attempted again. This error response is usually the result of a failed authentication attempt or a file transfer that could not complete.
500-599	Result codes in this range indicate that the requested action did not take place and the failure is permanent. The client should not attempt to send the command again. This error response is usually the result of an invalid command name, a syntax error or the client not having the appropriate access rights to a resource.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpSendResponse](#)

FtpGetCommandUsage Function

```
UINT WINAPI FtpGetCommandUsage(  
    HSERVER hServer,  
    LPCTSTR LpszCommand  
);
```

Return the number of times a specific command has been issued by all clients.

Parameters

hServer

The server handle.

lpszCommand

A pointer to a string that specifies a command name. The name is not case-sensitive, but must match a valid server command exactly. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is the number of times the command has been issued by all clients since the server was started. If the server handle or command name is invalid, or the command has never been issued, the return value will be zero.

Remarks

The command name provided to this function must match the commands defined in RFC 959 or related protocol standards. It is important to distinguish between commands recognized by an FTP server and the commands that client programs may use. For example, the standard Windows FTP command line program provides commands such as GET and PUT to download and upload files. However, those are not the actual commands sent to a server. Instead, the corresponding server commands issued by a client application would be RETR (retrieve) and STOR (store). Refer to [File Transfer Protocol Commands](#) for a complete list of server commands.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

See Also

[FtpEnableCommand](#), [FtpGetCommandLine](#), [FtpGetCommandResult](#)

FtpGetProgramExitCode Function

```
BOOL WINAPI FtpGetProgramExitCode(  
    HSERVER hServer,  
    UINT nClientId,  
    LPDWORD lpdwExitCode  
);
```

Return the exit code of the last program executed by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpdwExitCode

A pointer to an unsigned integer that will contain the program exit code when the function returns. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **FtpGetProgramExitCode** function returns the exit code of a registered program that was executed by the client using the SITE EXEC command. By convention, most programs return an exit code in the range of 0-255, with an exit code of zero indicating success. The exit code is commonly used by custom programs to communicate status information back to the server application.

Permission to use the SITE EXEC is not granted to authenticated users by default, and is limited to only those programs which are explicitly registered with the server. Exercise caution when allowing a client to execute a program on the server because this can expose the server to significant security risks. The programs that are registered for use with the SITE EXEC command should be thoroughly tested before being deployed on the server and should only be console programs that write to standard output.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftsv10.lib

See Also

[FtpGetProgramName](#), [FtpGetProgramOutput](#), [FtpRegisterProgram](#)

FtpGetProgramName Function

```
INT WINAPI FtpGetProgramName(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR lpszProgramName,  
    INT nMaxLength  
);
```

Return the name of the last program executed by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszProgramName

A pointer to a string buffer that will contain the name of the last program executed by the client. This parameter cannot be NULL and should be at least 32 characters in size.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This parameter must have a value greater than zero.

Return Value

If the function succeeds, the return value is the length of the command name. If the server handle and client ID do not specify a valid client session, the function will return zero. If the client has not executed any programs, this function will return zero.

Remarks

The **FtpGetProgramName** function returns the name of the last program that was executed by the client using the SITE EXEC command. The name that is returned is the alias assigned to the program, not the full path to the executable file. The server application would typically use this function in an event handler when processing the FTP_CLIENT_EXECUTE event to determine which program has been executed on behalf of the client. The **FtpGetProgramExitCode** function will return the program's exit code and the **FtpGetProgramOutput** function can be used to obtain a copy of the output generated by the program.

Permission to use the SITE EXEC is not granted to authenticated users by default, and is limited to only those programs which are explicitly registered with the server. Exercise caution when allowing a client to execute a program on the server because this can expose the server to significant security risks. The programs that are registered for use with the SITE EXEC command should be thoroughly tested before being deployed on the server and should only be console programs that write to standard output.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetProgramExitCode](#), [FtpGetProgramOutput](#), [FtpRegisterProgram](#)

FtpGetProgramOutput Function

```
DWORD WINAPI FtpGetProgramOutput(  
    HSERVER hServer,  
    UINT nClientId,  
    LPBYTE lpBuffer,  
    DWORD dwBufferSize  
);
```

Return a copy of the standard output from the last program executed by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpBuffer

A pointer to a buffer that will contain the output from the last program executed by the client. If this parameter is NULL, the function will return the number of bytes of data that was output by the program. Note that this output is not null terminated.

dwBufferSize

The maximum number of bytes that can be copied into the buffer. If the *lpBuffer* parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the number of bytes copied into the specified buffer. If the server handle and client ID do not specify a valid client session, the function will return zero. If the client has not executed any programs, the return value will be zero.

Remarks

The **FtpGetProgramOutput** function is used to obtain a copy of the output generated by the program executed using the SITE EXEC command. To determine the number of bytes of output available to read, call this function with the *lpBuffer* parameter as NULL and the *dwBufferSize* parameter with a value of zero. The return value will be the number of bytes of data that was output by the program. It should be noted that for Unicode builds, the buffer is a byte array, not an array of characters, and will not be null terminated.

This function returns the raw output from the command which may contain escape sequences, control characters and embedded nulls. When the application processes the output returned by this function, it should never coerce the buffer pointer to an LPTSTR value because there is no guarantee that the data will be null-terminated. To obtain the output from the command as a string, use the **FtpGetProgramText** function.

Example

```
LPBYTE lpBuffer = NULL; // A pointer to the output buffer  
DWORD cbBuffer = 0;     // Number of bytes in the output buffer  
  
// Determine the number of bytes in the output buffer  
cbBuffer = FtpGetProgramOutput(hServer, nClientId, NULL, 0);  
  
if (cbBuffer > 0)
```

```
{
    // Allocate memory for the buffer
    lpBuffer = (LPBYTE)LocalAlloc(LPTR, cbBuffer + 1);

    // Copy the program output to the buffer
    if (lpBuffer != NULL)
        cbBuffer = FtpGetProgramOutput(hServer, nClientId, lpBuffer, cbBuffer +
1);
}

// Free the memory allocated for the buffer when finished
if (lpBuffer != NULL)
{
    LocalFree((HLOCAL)lpBuffer);
    lpBuffer = NULL;
    cbBuffer = 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[FtpGetProgramExitCode](#), [FtpGetProgramName](#), [FtpGetProgramText](#), [FtpRegisterProgram](#)

FtpGetProgramText Function

```
INT WINAPI FtpGetProgramText(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR LpszBuffer,  
    INT nMaxLength  
);
```

Return a copy of the standard output from the last program in a string buffer.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszBuffer

A pointer to a buffer that will contain the output from the last program executed by the client as a string. If this parameter is NULL, the function will return the number of bytes of characters that was output by the program, not including a terminating null character.

nMaxLength

The maximum number of bytes that can be copied into the buffer. If the *lpszBuffer* parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the specified string buffer, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero. If the client has not executed any programs, the return value will be zero.

Remarks

The **FtpGetProgramText** function is used to obtain a copy of the output generated by the program executed using the SITE EXEC command. To determine the number of characters of output available to read, call this function with the *lpszBuffer* parameter as NULL and the *nMaxLength* parameter with a value of zero. The return value will be the number of characters that were output by the program. If the application dynamically allocates the string buffer, make sure that it allocates an extra character for the terminating null character.

This function will only return textual output from the command and any non-printable control characters and the escape character will be replaced with a space. To obtain the unfiltered output from the last command that was executed, use the **FtpGetProgramOutput** function.

Example

```
LPTSTR lpszBuffer = NULL; // A pointer to the output buffer  
INT cchBuffer = 0; // Number of bytes in the output buffer  
  
// Determine the number of bytes in the output buffer  
cchBuffer = FtpGetProgramText(hServer, nClientId, NULL, 0);  
  
if (cchBuffer > 0)  
{
```

```
// Allocate memory for the string buffer
lpszBuffer = (LPTSTR)LocalAlloc(LPTR, (cchBuffer + 1) * sizeof(TCHAR));

// Copy the program output to the buffer
if (lpszBuffer != NULL)
    cchBuffer = FtpGetProgramText(hServer, nClientId, lpszBuffer, cchBuffer
+ 1);
}

// Free the memory allocated for the buffer when finished
if (lpszBuffer != NULL)
{
    LocalFree((HLOCAL)lpszBuffer);
    lpszBuffer = NULL;
    cchBuffer = 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetProgramExitCode](#), [FtpGetProgramName](#), [FtpGetProgramOutput](#), [FtpRegisterProgram](#)

FtpGetRenamedFile Function

```
INT WINAPI FtpGetRenamedFile(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR lpszFileName,  
    INT nMaxLength  
);
```

Return the original name of a file being renamed by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A pointer to a string buffer that will contain the full path of the last file or directory that was renamed. It is recommended that this buffer be at least MAX_PATH characters in size. The value of this parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null characters. This parameter must have a value larger than zero.

Return Value

If the function succeeds, the return value is the length of the file or directory path, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero. If the client has not renamed a file or directory, this function will return zero.

Remarks

When a client wishes to rename a file or directory, it must send two commands in sequence to the server. The first command is RNFR (rename from) which specifies the original name of the file or directory to be renamed. The second command is RNTD (rename to) and must be sent immediately after the RNFR command and specifies the new name for the file or directory. The **FtpGetRenamedFile** function will return the full path of the local file or directory that was specified by the RNFR command. Typically this function is used by an event handler that processes the FTP_CLIENT_COMMAND event to determine the original path name.

This function is only guaranteed to return a meaningful value when called within the context of the FTP_CLIENT_COMMAND event. Calling this function outside of the event handler will return the path of the last renamed file, but there is no way to determine at what point the client issued the command to rename a file or directory. To obtain the new file or directory name, the **FtpGetCommandFile** function should be called from within the event handler.

The RNFR and RNTD commands can also be used to move a file or directory to a new location. For example, they could be used to move a file from one directory to another. An application should never make the assumption that the paths of the original and new file will be the same.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetCommandFile](#)

FtpGetServerAddress Function

```
INT WINAPI FtpGetServerAddress(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

Return the IP address of the server.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszAddress

A pointer to a string buffer that will contain the server IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the server handle or the client ID is invalid, or the buffer is not large enough to store the complete address, the function will return a value of zero.

Remarks

This function will return the IP address assigned to the specified server as a printable string. If the *nClientId* parameter has a value of zero, this function will return the IP address assigned to the local system. If the FTP_SERVER_NATROUTER option was specified when the server was started, this function will return the external IP address assigned to the system. If the *nClientId* parameter specifies a valid client session, this function will return the IP address that the client used to establish the connection with the server. To determine the IP address assigned to the client, use the **FtpGetClientAddress** function.

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetClientAddress](#), [FtpGetServerName](#), [FtpSetServerAddress](#)

FtpGetServerDirectory Function

```
INT WINAPI FtpGetServerDirectory(  
    HSERVER hServer,  
    LPTSTR lpszDirectory,  
    INT nMaxLength  
);
```

Return the full path to the root directory assigned to the specified server.

Parameters

hServer

The server handle.

lpszDirectory

A pointer to a string buffer that will contain the server root directory, terminated with a null character. It is recommended that this buffer be at least MAX_PATH characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle is invalid, or the buffer is not large enough to store the complete path, the function will return a value of zero.

Remarks

The **FtpGetServerDirectory** function will return the full path to the root directory assigned to the server instance. The root directory may be specified as part of the server configuration, or if no directory is specified by the application, a temporary root directory will be created and this function can be used to obtain the full path to the directory. When the application specifies a root directory, it may use environment variables such as %AppData% in the path. This function will return the fully resolved path name, with all environment variables expanded.

There is no corresponding function to change the server root directory after the server has started. To change the root directory, you must stop the server using the **FtpServerStop** function and then start another instance of the server with a configuration that specifies the new directory.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetServerAddress](#), [FtpGetServerIdentity](#), [FtpGetServerName](#)

FtpGetServerError Function

```
DWORD WINAPI FtpGetServerError(  
    HSERVER hServer,  
    LPTSTR lpszError,  
    INT nMaxLength  
);
```

Return the last server error code and a description of the error.

Parameters

hServer

The server handle.

lpszError

A pointer to a string buffer that will contain a description of the error. If the error description is not needed, this parameter may be NULL.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the error string buffer, including the terminating null character. If the *lpszError* parameter is NULL, this value should be zero.

Return Value

An unsigned integer value that specifies the last error that occurred. A value of zero indicates that there was no error.

Remarks

Error codes are unsigned 32-bit values which are private to each server. You should call the **FtpGetServerError** function immediately when a function's return value indicates that an error has occurred. That is because some functions clear the last error code when they succeed.

If the *hServer* parameter is specified with a value of INVALID_SERVER, this function will return the last error that occurred for the current thread. This value should only be used when the function does not have access to a valid server handle, such as when the **FtpServerStart** function fails.

It is important to note that the error codes returned by this function are different than the command result codes that are defined in RFC 959, the standard protocol specification for FTP. This function is used to determine reason that an API function has failed, and should not be used to determine if a command issued by the client was successful. The **FtpSendResponse** function is used to send result codes to the client, and the **FtpGetCommandResult** function can be used to determine the result of the last command sent by the client.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SERVER or FTP_ERROR. Those functions which clear the last error code when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetCommandResult](#), [FtpSendResponse](#)

FtpGetServerIdentity Function

```
INT WINAPI FtpGetServerIdentity(  
    HSERVER hServer,  
    LPTSTR lpszIdentity,  
    INT nMaxLength  
);
```

Return the identity of the specified server.

Parameters

hServer

The server handle.

lpszIdentity

A pointer to a string buffer that will contain the identity of the server when the function returns, terminated with a null character. This parameter cannot be NULL. It is recommended that this buffer be at least 32 characters in length.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle is invalid, or the buffer is not large enough to store the complete path, the function will return a value of zero.

Remarks

The **FtpGetServerIdentity** function returns the identity string that was specified as part of the server configuration. It is used for informational purposes only and does not affect the operation of the server. Typically the string specifies the name of the application and a version number, and is displayed whenever a client establishes its initial connection to the server. The **FtpSetServerIdentity** function can be used to change the identity string associated with the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetClientIdentity](#), [FtpSetClientIdentity](#), [FtpSetServerIdentity](#)

FtpGetServerLogFile Function

```
BOOL WINAPI FtpGetServerLogFile(  
    HSERVER hServer,  
    UINT * lpnLogFormat,  
    UINT * lpnLogLevel,  
    LPTSTR lpszFileName,  
    INT nMaxLength  
);
```

Return the current log file format and the full path to the file.

Parameters

hServer

The server handle.

lpnLogFormat

A pointer to an integer value that will contain the log file format being used when the function returns. If this information is not needed, this parameter may be NULL. The following formats are supported:

Constant	Description
FTP_LOGFILE_NONE (0)	This value specifies that the server should not create or update a log file.
FTP_LOGFILE_COMMON (1)	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about file transfers.
FTP_LOGFILE_EXTENDED (2)	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session.

lpnLogLevel

A pointer to an integer value that will contain the level of detail the server uses when generating the log file. The minimum value is 1 and the maximum value is 10. If this information is not needed, this parameter may be NULL.

lpszFileName

A pointer to a string buffer that will contain the full path to the log file. This parameter may be NULL if this information is not required.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the file name string, including the terminating null character. If the *lpszFileName* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the current log file format. Refer to the **FTPSEVERCONFIG** structure definition for a list of supported log file formats. If the server handle is invalid or logging has not been enabled, this function will return a value of zero.

Remarks

If the server is configured with logging enabled, but a log file name is not explicitly provided, then the server will automatically generate one. This function can be used to get the full path to the current log file along with the format that is being used to record client session data. Normally the log file is held open by the server thread while it is active, however you can call the **FtpRenameServerLogFile** function to explicitly rename or delete the log file.

To change the name of the log file, the log file format or level of detail, use the **FtpSetServerLogFile** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpRenameServerLogFile](#), [FtpSetServerLogFile](#)

FtpGetServerMemoryUsage Function

```
BOOL WINAPI FtpGetServerMemoryUsage(  
    HSERVER hServer,  
    ULARGE_INTEGER * LpMemUsage  
);
```

Return the amount of memory allocated for the server and all client sessions.

Parameters

hServer

The server handle.

lpMemUsage

A pointer to a ULARGE_INTEGER variable which will specify how much memory has been allocated by the server. This parameter will be initialized to a value of zero by the function and updated with the total number of bytes allocated by the server and all active client sessions when it returns.

Return Value

If the function succeeds, the return value is non-zero and the memory usage value will be updated. If the server handle is invalid, or the server cannot be locked, the return value is zero. Call the **FtpGetServerError** function to determine the cause of the failure.

Remarks

This function returns the amount of memory allocated by the server and all active client sessions. It enumerates all of memory allocations made by the server process and client session threads and returns the total number of bytes allocated for the server process. This value reflects the amount of memory explicitly allocated by this library and does not reflect the total working set size of the process, or memory allocated by any other libraries. To determine the working set size for the process, refer to the Win32 **GetProcessWorkingSetSize** and **GetProcessMemoryInfo** functions.

This function forces the server into a locked state, and all client sessions will block until the function returns. Because this function enumerates all heaps allocated for the server process, it can be an expensive operation, particularly when there are a large number of active clients connected to the server. Frequent use of this function can significantly degrade the performance of the server. It is primarily intended for use as a debugging tool to determine if memory usage is the result of an increase in active client sessions. If the value returned by the function remains reasonably constant, but the amount of memory allocated for the process continues to grow, it could indicate a memory leak in some other area of the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetServerStackSize](#), [FtpSetServerStackSize](#)

FtpGetServerName Function

```
INT WINAPI FtpGetServerName(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR lpszHostName,  
    INT nMaxLength  
);
```

Return the host name assigned to the specified server.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszHostName

A pointer to a string buffer that will contain the server host name, terminated with a null character. It is recommended that this buffer be at least 64 characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the server handle or the client ID is invalid, or the buffer is not large enough to store the complete hostname, the function will return a value of zero.

Remarks

This function will return the host name assigned to the specified server. If the *nClientId* parameter has a value of zero, the function will return the default host name that was specified as part of the server configuration. If no host name was explicitly assigned to the server, then it will return the local system name. If the *nClientId* parameter specifies a client session, then it this function will return the host name that the client used to establish the connection. If the client sends the HOST command to the server, this function will return the host name provided by the client.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetServerAddress](#)

FtpGetServerPriority Function

```
INT WINAPI FtpGetServerPriority(  
    HSERVER hServer  
);
```

Return the current priority assigned to the specified server.

Parameters

hServer

The server handle.

Return Value

If the function succeeds, the return value is the priority for the specified server. If the function fails, the return value is FTP_PRIORITY_INVALID. To get extended error information, call **FtpGetServerError**.

Remarks

The **FtpGetServerPriority** function can be used to determine the current priority assigned to the server. It will return one of the following values:

Constant	Description
FTP_PRIORITY_BACKGROUND (0)	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
FTP_PRIORITY_LOW (1)	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
FTP_PRIORITY_NORMAL (2)	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
FTP_PRIORITY_HIGH (3)	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
FTP_PRIORITY_CRITICAL (4)	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cstools10.h.

Import Library: csftsv10.lib

See Also

[FtpServerStart](#), [FtpSetServerPriority](#)

FtpGetServerStackSize Function

```
DWORD WINAPI FtpGetServerStackSize(  
    SOCKET hServer  
);
```

Return the initial size of the stack allocated for threads created by the server.

Parameters

hServer

The server handle.

Return Value

If the function succeeds, the return value is the amount of memory that will be allocated for the stack in bytes. If the function fails, the return value is zero. To get extended error information, call **FtpGetServerError**.

Remarks

The **FtpGetServerStackSize** function returns the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `csftools10.h`.

Import Library: `csftsv10.lib`

See Also

[FtpGetServerMemoryUsage](#), [FtpServerStart](#), [FtpSetServerStackSize](#)

FtpGetServerTransferInfo Function

```
BOOL WINAPI FtpGetServerTransferInfo(  
    HSERVER hServer,  
    UINT nClientId,  
    LPFTPSEVERTRANSFER lpTransferInfo  
);
```

Return information about the file transfer that was performed by the server for the specified client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpTransferInfo

A pointer to an **FTPSEVERTRANSFER** structure that will contain information about the last file transfer. This parameter cannot be NULL, and the *dwSize* member of the structure must be initialized to specify the structure size prior to calling this function.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero. This function should only be called after the client has issued the APPE, RETR, STOR or STOU commands to initiate a file transfer, otherwise the return value will be zero.

Remarks

The **FtpGetServerTransferInfo** function is used to obtain information about the last file transfer that was performed by the server for the specified client. This function is typically called within an event handler to determine how many bytes of data were transferred, the type of file and the full path to the file on the local system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetCommandResult](#), [FTPSEVERTRANSFER](#)

FtpGetServerUuid Function

```
INT WINAPI FtpGetServerUuid(  
    HSERVER hServer,  
    UUID * LpUuid  
);
```

Return the UUID assigned to the specified server.

Parameters

hServer

The server handle.

lpUuid

A pointer to a UUID structure that will contain the server UUID when the function returns. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If either the server handle is invalid or the pointer to the UUID structure is NULL, the function will return a value of zero.

Remarks

The **FtpGetServerUuid** function returns the Universally Unique Identifier (UUID) that has been assigned to the server. The UUID may either be generated by the application and assigned as part of the server configuration, or an ephemeral UUID may be automatically generated when the server is started. To obtain a printable string version of the UUID, use the **FtpGetServerUuidString** function.

There is no corresponding function to change the UUID assigned to an active server. The server UUID is assigned when the server is started, and it must be a unique value that is maintained throughout the lifetime of the server. To change the UUID associated with the server, the server must be stopped using the **FtpServerStop** function and another instance of the server started with the new UUID.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[FtpGetServerUuidString](#), [FTPSEVERCONFIG](#)

FtpGetServerUuidString Function

```
INT WINAPI FtpGetServerUuidString(  
    HSERVER hServer,  
    LPTSTR lpszHostUuid,  
    INT nMaxLength  
);
```

Return the UUID assigned to the server as a printable string.

Parameters

hServer

The server handle.

lpszHostUuid

A pointer to a string buffer that will contain the server UUID, terminated with a null character. It is recommended that this buffer be at least 40 characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the server handle or the buffer is not large enough to store the complete UUID string, the function will return a value of zero.

Remarks

The **FtpGetServerUuidString** function returns the Universally Unique Identifier (UUID) that has been assigned to the server. The UUID may either be generated by the application and assigned as part of the server configuration, or an ephemeral UUID may be automatically generated when the server is started. To obtain the numeric UUID value, use the **FtpGetServerUuid** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetServerUuid](#), [FTPSERVERCONFIG](#)

FtpIsClientAnonymous Function

```
BOOL WINAPI FtpIsClientAnonymous(  
    HSERVER hServer,  
    UINT nClientId  
);
```

Determine if the specified client has authenticated as an anonymous user.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the client session has been authenticated, this function will return a non-zero value, otherwise it will return zero. If the server handle and client ID are valid, and the client session has been authenticated, this function will clear the last error code.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[FtpAuthenticateClient](#), [FtpGetClientCredentials](#), [FtpIsClientAuthenticated](#)

FtpIsClientAuthenticated Function

```
BOOL WINAPI FtpIsClientAuthenticated(  
    HSERVER hServer,  
    UINT nClientId  
);
```

Determine if the specified client session has been authenticated.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the client session has been authenticated, this function will return a non-zero value, otherwise it will return zero. If the server handle and client ID are valid, this function will clear the last error code.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[FtpAuthenticateClient](#), [FtpGetClientCredentials](#), [FtpIsClientAnonymous](#)

FtpIsCommandEnabled Function

```
BOOL WINAPI FtpIsCommandEnabled(  
    HSERVER hServer,  
    LPCTSTR LpszCommand  
);
```

Determine if a specific server command has been enabled or disabled.

Parameters

hServer

The server handle.

lpszCommand

A pointer to a NULL terminated string that specifies the name of the command. The command name is not case-sensitive, but the value must otherwise match the exact command name.

Partial matches are not recognized by this function. This parameter cannot be NULL.

Return Value

If the command is enabled, this function will return a non-zero value. If the command is disabled, the server handle is invalid or the command name does not match a supported command, this function will return zero.

Remarks

The **FtpIsCommandEnabled** function is used to determine whether a specific command is enabled. Typically this function is used in an event handler to make sure the command issued by a client is recognized by the server and enabled for use. Commands can be enabled and disabled using the **FtpEnableCommand** function.

This function does not account for the permissions granted to a specific client session. Clients are assigned access rights when they are authenticated using the **FtpAuthenticateClient** function, and certain commands can be limited by the permissions granted to the client. For example, even though the STOR command is enabled, a client must have the FTP_ACCESS_WRITE permission to use the command to upload a file to the server. For a list of access rights, see [User Access Constants](#).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpAuthenticateClient](#), [FtpEnableCommand](#), [FtpGetCommandName](#)

FtpRegisterProgram Function

```
BOOL WINAPI FtpRegisterProgram(  
    HSERVER hServer,  
    UINT nHostId,  
    LPCTSTR lpszCommandName,  
    LPCTSTR lpszProgramFile,  
    LPCTSTR lpszParameters,  
    LPCTSTR lpszDirectory  
);
```

Register a program for use with the SITE EXEC command.

Parameters

hServer

The server handle.

nHostId

An unsigned integer that identifies the virtual host associated with the program. This value should always be zero.

lpszCommandName

A pointer to a string which specifies the name of the site specific command. This is the name that is passed to the SITE EXEC command and does not need to match the actual name of the executable file on the local system. The maximum length of the command name is 32 characters. This parameter cannot be NULL.

lpszProgramFile

A pointer to a string that specifies the full path to the executable program. This parameter cannot be NULL.

lpszParameters

A pointer to a string that specifies additional parameters for the program. If the program does not require any command line parameters, this value may be NULL or point to an empty string.

lpszDirectory

A pointer to a string that specifies the current working directory for the program. If this parameter is NULL or points to an empty string, the server will use the current working directory of the client that issues the SITE EXEC command.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **FtpRegisterProgram** function registers an executable program for use with the SITE EXEC command. Because this can present a significant security risk to the server, clients are not given permission to use this command by default. A client must be explicitly granted permission to use SITE EXEC by including FTP_ACCESS_EXECUTE as one of the permissions when authenticating the client session with the **FtpAuthenticateClient** function or creating a virtual user using the **FtpAddVirtualUser** function.

To give the server complete control over what programs can be executed using SITE EXEC, the program must be registered with the server and referenced by an alias specified by the

lpszCommandName parameter. The maximum length of a program name is 31 characters and it must be at least 3 characters in length. The name must only consist of alphanumeric characters and the first character of the program name cannot be numeric. The program name is not case-sensitive, however convention is to use upper-case characters. If a program name is specified that already has been registered, it will be updated with the new information provided by this function.

The *lpszProgramFile* string specifies file name of the program that will be executed. You should not install any executable programs in the server root directory or its subdirectories. A client should never have the ability to directly access the executable file itself. It is permitted to have multiple command names that reference the same executable file. The only requirement is that the command names be unique. The program name may contain environment variables surrounded by % symbols. For example, %ProgramFiles% would be expanded to the **C:\Program Files** folder.

It is important to note that the program specified by *lpszProgramFile* must be an executable file, not a script or batch file. If the program name does not contain a directory path, then the standard Windows pathing rules will be used when searching for an executable file that matches the given name. It is recommended that you always provide a full path to the executable file.

The *lpszParameters* string is used to define optional command line parameters that will be included with the command. This string can contain placeholders that are replaced by additional parameters specified by the client when it sends the SITE EXEC command. First replacement parameter is %1, the second is %2 and so on.

The executable program that is registered using this function must be a console application that writes to standard output. Programs that write directly to a console, or programs written to use a Windows user interface are not supported and will yield unpredictable results. In most cases, those programs that do not use standard input and output will be forcibly terminated by the server. If the program attempts to read from standard input, it will immediately encounter an end-of-file condition. Programs executed by the SITE EXEC command have no input; it is similar to a program that has its input redirected from the NUL: device. If the program must process a file on the server, the local file name should be passed as a command line parameter.

The output from the program will be redirected back to the client control channel. The output should be textual, with each line of text terminated by a carriage return and linefeed (CRLF). Programs that write binary data to standard output, particular data with embedded nulls, will yield unpredictable results and are not supported. To ensure that the program output conforms to the protocol standard, any non-printable characters will be replaced with a space and each line of output will be prefixed by a single space. The server application can obtain a copy of the output from the last command by calling the **FtpGetProgramOutput** function.

If the server is running on a system with User Account Control (UAC) enabled and does not have elevated privileges, do not register a program that requires elevated privileges or has a manifest that specifies the requestedExecutionLevel as requiring administrative privileges.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetProgramExitCode](#), [FtpGetProgramName](#), [FtpGetProgramOutput](#)

FtpRenameServerLogFile Function

```
BOOL WINAPI FtpRenameServerLogFile(  
    HSERVER hServer,  
    LPCTSTR lpszFileName  
);
```

Rename or delete the current log file being updated by the server.

Parameters

hServer

The server handle.

lpszFileName

A pointer to a string that specifies the file name the current log file should be renamed to. If this parameter is NULL or an empty string, the current log file will be deleted.

Return Value

If the function succeeds, the return value is non-zero. If the server handle does not specify a valid server, the function will return zero. If logging is not currently enabled for the server, this function will return zero.

Remarks

The **FtpRenameServerLogFile** function is used to rename or delete the current log file. Note that this does not change the current log file name or disable logging by the server. It only changes the file name of the current log file, or removes the log file if the *lpszFileName* parameter is NULL. This can be useful if you want your server to perform log file rotation, archiving the current log file. By renaming the current log file, the server will automatically create a new log file with original file name.

This function must be used to rename or delete the current log file while logging is active because the server holds an open handle on the file. The application should not use the **FtpGetServerLogFile** function to obtain the log file name and then use the **MoveFileEx** or **DeleteFile** functions with that file.

To disable logging, use the **FtpSetServerLogFile** function and specify the logging format as FTP_LOGFILE_NONE.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetServerLogFile](#), [FtpSetServerLogFile](#)

FtpSendResponse Function

```
BOOL WINAPI FtpSendResponse(  
    HSERVER hServer,  
    UINT nClientId,  
    UINT nResultCode,  
    LPCTSTR lpszMessage  
);
```

Send a result code and message to the client in response to a command.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

nResultCode

An unsigned integer value which specifies the result code.

lpszMessage

A pointer to a string which specifies a message to be sent to the client. If this parameter is NULL or points to an empty string, a default message associated with the result code will be used.

Return Value

If the result code and message text was sent to the client, the return value is non-zero. If the server handle and client ID do not specify a valid client session, or the result code is invalid, this function will return zero.

Remarks

The **FtpSendResponse** function is used to respond to a command issued by the client. Command responses are normally handled by the server as a normal part of processing a command and this function is only used if the application has implemented custom commands or wishes to modify the standard responses sent by the server. The message may be a maximum of 2048 characters and may include embedded carriage-return and linefeed characters. If no message is specified, then a default message will be sent based on the result code.

Result codes must be three digits (in the range of 100 through 999) and although this function will support the use of non-standard result codes, it is recommended that the client application use the standard codes defined in RFC 959 whenever possible. The use of non-standard result codes may cause problems with FTP clients that expect specific result codes in response to a particular command. For more information, refer to the **FtpGetCommandResult** function.

This function should only be called once in response to a command sent by the client. If a result code has already been sent in response to a command and this function is called, it will fail and return a value of zero. This is necessary because sending multiple result codes in response to a single command may cause unpredictable behavior by the client.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetCommandResult](#)

FtpServerAsyncNotify Function

```
BOOL WINAPI FtpServerAsyncNotify(  
    HSERVER hServer,  
    HWND hWnd,  
    UINT uMsg  
);
```

Enable or disable asynchronous notification of changes in server status.

Parameters

hServer

The server handle.

hWnd

A handle to the window whose window procedure will receive the notification message.

uMsg

The user-defined message that will be sent to the notification window.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call the **FtpGetServerError** function.

Remarks

The **FtpServerAsyncNotify** function is used by an application to enable or disable asynchronous notifications. The message window is typically the main UI window and these notifications are used signal to the application that it should update the user interface. If the *hWnd* parameter is not NULL, it must specify a valid window handle and the user-defined message must have a value of **WM_USER** or higher. The application cannot specify a notification message that is reserved by the operating system. The pseudo-handle **HWND_BROADCAST** cannot be specified as the notification window. If the *hWnd* parameter is NULL, notifications for the specified server will be disabled.

When asynchronous notifications are enabled for a server, the server will post the user-defined message to the window whenever there is a change in status or after a client has connected or disconnected from the server. The *wParam* message parameter will contain the notification message and the *lParam* message parameter will contain the handle to the server or the client ID. The following notification messages are defined:

Constant	Description
FTP_NOTIFY_STARTUP	This notification is sent when the server has started and is preparing to accept client connections. This notification is only sent once, and only if asynchronous notifications are enabled immediately after the FtpServerStart function is called. This message will not be sent once the server has begun accepting client connections or when notification messages are disabled and then subsequently re-enabled at a later time. The <i>lParam</i> message parameter will specify the handle to the server.
FTP_NOTIFY_LISTEN	This notification is sent when the server is listening for client connections. This notification message may be sent

	to the application multiple times over the lifetime of the server. If the server was suspended, this notification will be sent after the application calls the FtpServerResume function to resume accepting client connections. The <i>lParam</i> message parameter will specify the handle to the server.
FTP_NOTIFY_SUSPEND	This notification is sent when the server suspends accepting new connections because the application has called the FtpServerSuspend function. This notification message may be sent to the application multiple times over the lifetime of the server. The <i>lParam</i> message parameter will specify the handle to the server.
FTP_NOTIFY_RESTART	This notification is sent when the server is restarted using the FtpServerRestart function. Note that the server socket handle provided by the <i>lParam</i> message parameter will specify the new socket handle of the restarted server instance, not the original socket handle. The <i>lParam</i> message parameter will specify the handle to the server.
FTP_NOTIFY_CONNECT	This notification is sent when the server accepts a client connection and the thread that manages the client session has begun processing network events for that client. This message notification will not be sent if the client connection is rejected by the server. The <i>lParam</i> message parameter will specify the unique ID of the client that connected to the server.
FTP_NOTIFY_DISCONNECT	This notification is sent when the client disconnects from the server and the client socket has been closed. This notification message may not occur for each client session that is forced to terminate as the result of the server being stopped using the FtpServerStop function. The <i>lParam</i> message parameter will specify the unique ID of the client that disconnected from the server.
FTP_NOTIFY_SHUTDOWN	This notification is sent when the server thread is in the process of terminating. At the time the application processes this notification message, the server handle in <i>lParam</i> will reference the defunct server and cannot be used with other server functions. The <i>lParam</i> message parameter will specify the handle to the server.

If asynchronous notifications are enabled, you should never use those notifications as a replacement for an event handler. When an event occurs, the callback function that handles the event is invoked in the context of the thread that manages the client session. The application should exchange data with the client within that event handler and not in response to a notification message. These notification messages should only be used to update the application UI in response to changes in the status of the server.

The FTP_NOTIFY_CONNECT and FTP_NOTIFY_DISCONNECT notifications are different from the other server notifications because the *lParam* message parameter does not specify the server

handle, but rather the unique client ID associated with the session that connected to or disconnected from the server. Use the **FtpGetClientServer** function to obtain a handle to the server that created the client session. Note that at the time the application processes the FTP_NOTIFY_DISCONNECT notification message, the client session will have already terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cstools10.h.

Import Library: csftsv10.lib

See Also

[FtpGetClientServer](#), [FtpServerStart](#)

FtpServerDisableTrace Function

```
BOOL WINAPI FtpServerDisableTrace();
```

Disable the logging of network function calls.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetServerError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[FtpServerEnableTrace](#)

FtpServerEnableTrace Function

```
BOOL WINAPI FtpServerEnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

Enable the logging of network function calls to a file.

Parameters

lpszTraceFile

A pointer to a string that specifies the name of the log file. If this parameter is NULL or points to an empty string, a log file is created in the temporary directory for the current user.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_DEFAULT (0)	All function calls are written to the trace file. This is the default value.
TRACE_ERROR (1)	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING (2)	Only those function calls which fail, or return values which indicate a warning, are recorded in the trace file.
TRACE_HEXDUMP (4)	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call [FtpGetServerError](#).

Remarks

When trace logging is enabled, the log file is opened, appended to and closed for each socket function call. Using the same file name, you can do the same in your application to add additional information to the file if needed. This can provide an application-level context for the entries made by the library. Make sure that the file is closed after the data has been written. If a file name is not specified by the caller, a file named **cstrace.log** will be created in the temporary directory for the current user.

The TRACE_HEXDUMP option can produce very large files, since all data that is being sent and received by the application is logged. To reduce the size of the file, you can enable and disable logging around limited sections of code that you wish to analyze.

To redistribute an application that includes this debug logging functionality, the **cstrcv10.dll** library must be included as part of the installation package. This library provides the trace logging features, and if it is not available the **FtpServerEnableTrace** function will fail. Note that this is a standard Windows DLL and does not need to be registered, it only needs to be redistributed with your application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpServerDisableTrace](#)

FtpServerInitialize Function

```
BOOL WINAPI FtpServerInitialize(  
    LPCTSTR lpszLicenseKey,  
    LPINITDATA lpData  
);
```

The **FtpServerInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpData

Pointer to an INITDATA data structure. This parameter may be NULL if the initialization data for the library is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetServerError**. All other functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must insure that this function is called with a valid license key.

If the *lpData* argument is specified, it must point to an INITDATA structure which has been initialized by setting the *dwSize* member to the size of the structure. All other structure members should be set to zero. If the function is successful, then the INITDATA structure will be filled with identifying information about the library.

Although it is only required that **FtpServerInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **FtpServerUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **FtpServerInitialize** or **FtpServerUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpServerStart](#), [FtpServerStop](#), [FtpServerUninitialize](#), [INITDATA](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

FtpServerProc Function

```
VOID CALLBACK FtpServerProc(  
    HSERVER hServer,  
    UINT nClientId,  
    UINT nEventId,  
    DWORD dwError,  
    DWORD_PTR dwParam  
);
```

The **FtpServerProc** function is an application-defined callback function that processes events generated by the server process.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client that has issued a request to the server.

nEventId

An unsigned integer which specifies which event occurred. For a list of events, see [Server Event Constants](#).

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero. The **FtpGetServerError** function can be used to obtain additional information about the error code.

dwParam

A user-defined integer value which was specified when the server was started. This value is guaranteed to be large enough to store a pointer or handle value on both 32-bit and 64-bit platforms.

Return Value

None.

Remarks

The FTP_CLIENT_COMMAND event can be used to filter commands issued by the client. If the command performs an operation on a file or directory, the **FtpGetCommandFile** function can be used to obtain the file name on the server that the client specified. It is possible for the application to change the file name using the **FtpSetCommandFile** function to direct the server to use a different file than the one specified by the client. If the file name is changed and the event handler returns a value of zero, the server will perform the default action for the command using the new file name.

The FTP_CLIENT_TIMEOUT event will only occur for authenticated client sessions. The server has a second, internal timer that is maintained for unauthenticated sessions that requires the client to successfully authenticate itself within a fixed period of time. If a client does not authenticate within the allowed time period, the server will automatically close the control channel. For more information, refer to the documentation for the *nAuthTime* member of the **FTP_SERVER_CONFIG** structure.

The callback function will be called in the context of the thread that is currently managing the

client session. You must ensure that any access to global or static variables are synchronized, otherwise the results may be unpredictable. It is recommended that you do not declare any static variables within the callback function itself.

If the application has a graphical user interface, you should never attempt to directly modify a UI control from within an event handler. Controls should only be modified by the same UI thread that created their window. To change the user interface in response to a server event, use the **FtpServerAsyncNotify** function to enable asynchronous notifications and update the UI in response to the notification message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[FtpServerAsyncNotify](#), [FtpServerStart](#)

FtpServerRestart Function

```
HSERVER WINAPI FtpServerRestart(  
    HSERVER hServer  
);
```

Restart the server, terminating all active client sessions.

Parameters

hServer

The server handle.

Return Value

If the function succeeds, the return value is the new handle for the specified server. If the function fails, the return value is `INVALID_SERVER`. To get extended error information, call

FtpGetServerError.

Remarks

The **FtpServerRestart** function will restart the specified server, terminating all active client sessions. The server handle that is returned by the function is the handle for the new server instance, and the old handle value is no longer valid. If the function is unable to restart the server for any reason, the server thread is terminated. The server retains all of the configuration parameters from the previous instance, however the statistical information (such as the number of clients, files transferred, etc.) will be reset.

If an application calls this function from within an event handler, the active client session (the client for which the event handler was invoked) may not get a disconnect notification. It is recommended that this function only be called by the same thread that created the server using the **FtpServerStart** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csftsv10.lib`

See Also

[FtpServerStart](#), [FtpServerStop](#)

FtpServerResume Function

```
BOOL WINAPI FtpServerResume(  
    HSERVER hServer  
);
```

Resume accepting client connections.

Parameters

hServer

Handle to the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetServerError**.

Remarks

The **FtpServerResume** function instructs the server to resume accepting new client connections after the **FtpServerSuspend** function has been called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csftsv10.lib

See Also

[FtpServerRestart](#), [FtpServerStart](#), [FtpServerStop](#), [FtpServerSuspend](#), [FtpServerThrottle](#)

FtpServerStart Function

```
HSERVER WINAPI FtpServerStart(  
    LPCTSTR lpszLocalHost,  
    UINT nLocalPort,  
    DWORD dwOptions,  
    LPFTPSEVERCONFIG lpServerConfig,  
    FTPSERVPROC lpServerProc,  
    DWORD_PTR dwServerParam,  
    LPSECURITYCREDENTIALS lpCredentials  
);
```

The **FtpServerStart** function begins listening for client connections on the specified local address and port number. The server is started in its own thread and manages the client sessions independently of the calling thread. All interaction with the server and its client sessions takes place inside the callback function specified by the caller.

Parameters

lpszLocalHost

A pointer to a string which specifies the local hostname or IP address address that the server should be bound to. If this parameter is NULL or an empty string, then an appropriate address will automatically be used. If a specific address is used, the server will only accept client connections on the network interface that is bound to that address.

nLocalPort

The port number the server should use to listen for client connections. If a value of zero is specified, the server will use the standard port number 21 to listen for connections, or port 990 if the server is configured to use implicit SSL. The port number used by the application must be unique and multiple instances of a server cannot use the same port number. It is recommended that a port number greater than 5000 be used for private, application-specific implementations.

dwOptions

An unsigned integer value that specifies the options used when creating an instance of the server. For a list of options, see [Server Option Constants](#).

lpServerConfig

A pointer to an **FTPSEVERCONFIG** structure that specifies the configuration options for the server. If this parameter is NULL, then default configuration parameters will be used that starts the server in a restrictive mode.

lpServerProc

Specifies the address of the application defined callback function. For more information about the callback function, see the description of the **FtpServerProc** callback function. If this parameter is NULL, a default internal handler is used to process client commands.

dwServerParam

A user-defined integer value that is passed to the callback function. If the *lpServerProc* parameter is NULL, this value should be zero. This value is guaranteed to be large enough to store a pointer or handle value on both 32-bit and 64-bit platforms.

lpCredentials

Pointer to a **SECURITYCREDENTIALS** structure. If this parameter is NULL, the default security credentials for the server host name will be used. If security is enabled for the server, it is recommended that you provide a pointer to this structure with specific information about the

server certificate that should be used. If the security options are not specified in the server configuration, this parameter is ignored.

Return Value

If the function succeeds, the return value is a handle to a server session. If the function fails, the return value is `INVALID_SERVER`. To get extended error information, call **FtpGetServerError**.

Remarks

In most cases, the *lpzLocalHost* parameter should be a NULL pointer or an empty string. On a multihomed system, this will enable the server to accept connections on any appropriately configured network adapter. Specifying a hostname or IP address will limit client connections to that particular address. Note that the hostname or address must be one that is assigned to the local system, otherwise an error will occur.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the function will fail.

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

The handle returned by this function references the listening socket that was created when the server was started. The service is managed in another thread, and all interaction with the server and active client connections are performed inside the event handler. To disconnect all active connections, close the listening socket and terminate the server thread, call the **FtpServerStop** function.

The host UUID that is defined as part of the server configuration should be generated using the **uuidgen** utility that is included with the Windows SDK. You should not use the UUID that is provided in the example code, it is for demonstration purposes only. If no host UUID is specified in the server configuration, an ephemeral UUID will be generated automatically when the server is started.

If the server is started with security enabled, the *lpCredentials* parameter should be used to provide the server certificate information in a **SECURITYCREDENTIALS** structure. If the pointer is NULL, this function will search for a certificate that matches the hostname provided as part of the server configuration. This requires that the server certificate be installed in the personal certificate store of the current process user, and it must have a private key associated with it. To use a certificate with a different name, or one that is stored in a PFX file, a **SECURITYCREDENTIALS** structure must be defined and passed to this function.

Example

```
HSERVER hServer;
FTPSEVERCONFIG ftpConfig;

// Initialize the server configuration
ZeroMemory(&ftpConfig, sizeof(FTPSEVERCONFIG));
ftpConfig.dwSize = sizeof(ftpConfig);
ftpConfig.nMaxClients = 20;
ftpConfig.nMaxGuests = 5;
ftpConfig.nLogFormat = FTP_LOGFILE_EXTENDED;
ftpConfig.nAuthTime = 30;
```

```
ftpConfig.nIdleTime = 120;
ftpConfig.lpszIdentity = _T("MyProgram");
ftpConfig.lpszHostName = _T("server.company.com");
ftpConfig.lpszHostUuid = _T("10000000-1000-1000-1000-100000000000");
ftpConfig.lpszDirectory = _T("%ProgramData%\MyProgram\Files");
ftpConfig.lpszLogFile = _T("%ProgramData%\MyProgram\Server.log");

// Start the server
hServer = FtpServerStart(lpszLocalHost,
                        FTP_PORT_DEFAULT,
                        FTP_SERVER_LOCALUSER | FTP_SERVER_UNIXMODE,
                        &ftpConfig,
                        lpEventHandler,
                        0,
                        NULL);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpEnumServerClients](#), [FtpServerProc](#), [FtpServerRestart](#), [FtpServerStop](#), [FTPSEVERCONFIG](#), [SECURITYCREDENTIALS](#)

FtpServerStop Function

```
BOOL WINAPI FtpServerStop(  
    HSERVER hServer  
);
```

Stop the server, terminating all active client sessions and releasing the resources that were allocated for the server.

Parameters

hServer

The server handle.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid, the function will return a value of zero.

Remarks

The **FtpServerStop** function instructs the server to stop accepting client connections, disconnects all active client connections and terminates the thread that is managing the server session. The handle is no longer valid after the server has been stopped and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new server is started. An application should always consider the server handle to be opaque and never depend on it being a specific value.

If an application calls this function from within an event handler, the active client session (the client for which the event handler was invoked) may not get a disconnect notification. It is recommended that this function only be called by the same thread that created the server using the **FtpServerStart** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[FtpServerRestart](#), [FtpServerStart](#)

FtpServerSuspend Function

```
BOOL WINAPI FtpServerSuspend(  
    HSERVER hServer  
);
```

Suspend the server and reject new client connections.

Parameters

hServer

Handle to the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetServerError**.

Remarks

The **FtpServerSuspend** function instructs the server to suspend accepting new client connections. Any incoming client connections will be rejected with an error message indicating that the server is currently unavailable. To resume accepting client connections, call the **FtpServerResume** function. Suspending the server will have no effect on clients that have already established a connection with the server.

It is recommended that you only suspend a server if absolutely necessary, and only for brief periods of time. If you want to limit the number of active client connections or control the connection rate for clients, use the **FtpServerThrottle** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

See Also

[FtpServerRestart](#), [FtpServerResume](#), [FtpServerStart](#), [FtpServerStop](#), [FtpServerThrottle](#)

FtpServerThrottle Function

```
BOOL WINAPI FtpServerThrottle(  
    HSERVER hServer,  
    UINT nMaxClients,  
    UINT nMaxClientsPerAddress,  
    UINT nMaxGuests,  
    DWORD dwConnectionRate  
);
```

The **FtpServerThrottle** function limits the number of active client connections, connections per address and connection rate.

Parameters

hServer

Handle to the server.

nMaxClients

An integer value which specifies the maximum number of clients that may connect to the server. A value of zero specifies that there is no fixed limit to the number of client connections. A value of -1 specifies that the maximum number of clients should not be changed.

nMaxClientsPerAddress

An integer value which specifies the maximum number of clients that may connect to the server from the same IP address. A value of zero specifies that there is no fixed limit to the number of client connections per address. By default, there is a limit of four client connections per address. A value of -1 specifies that the maximum number of clients should not be changed.

nMaxGuests

An integer value which specifies the maximum number of guest users that may login to the server. A value of zero disables guest logins and requires that all clients authenticate with a valid username and password. A value of -1 specifies that the maximum number of guest users should not be changed.

dwConnectionRate

A value which specifies a restriction on the rate of client connections, limiting the number of connections that will be accepted within that period of time. A value of zero specifies that there is no restriction on the rate of client connections. The higher this value, the fewer the number of connections that will be accepted within a specific period of time. By default, there is no limit on the client connection rate. A value of -1 specifies that the connection rate should not be changed.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetServerError**.

Remarks

The **FtpServerThrottle** function is used to limit the number of connections and the connection rate to minimize the potential impact of a large number of client connections over a short period of time. This can be used to protect the server from a client application that is malfunctioning or a deliberate denial-of-service attack in which the attacker attempts to flood the server with connection attempts.

If the maximum number of client connections or maximum number of connections per address is exceeded, the server will reject subsequent connection attempts until the number of active client sessions drops below the specified threshold. Note that adjusting these values lower than the current connection limits will not affect clients that have already connected to the server. For example, if the **FtpServerStart** function is called with the maximum number of clients set to 100, and then **FtpServerThrottle** is called lowering that value to 75, no existing client connections will be affected by the change. However, the server will not accept any new connections until the number of active clients drops below 75.

Increasing the connection rate value will force the server to slow down the rate at which it will accept incoming client connection requests. For example, setting this parameter to a value of 1000 would limit the server to accepting one client connection every second, while a value of 250 would allow the server to accept four client connections per second. Note that significantly increasing the amount of time the server must wait to accept client connections can exceed the connection backlog queue, resulting in client connections being rejected.

It is recommended that you always specify conservative connection limits for your server application based on expected usage. Allowing an unlimited number of client connections can potentially expose the system to denial-of-service attacks and should never be done for servers that are accessible over the Internet.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[FtpServerRestart](#), [FtpServerResume](#), [FtpServerStart](#), [FtpServerSuspend](#)

FtpServerUninitialize Function

```
VOID WINAPI FtpServerUninitialize();
```

The **FtpServerUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **FtpServerInitialize** call before it can call any of the other library functions. When it has completed the use of library, the application must call **FtpServerUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There must be a call to **FtpServerUninitialize** for every successful call to **FtpServerInitialize** made by a process. Operations for all threads in the server are terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpServerInitialize](#), [FtpServerStart](#), [FtpServerStop](#)

FtpSetClientAccess Function

```
BOOL WINAPI FtpSetClientAccess(  
    HSERVER hServer,  
    UINT nClientId,  
    DWORD dwUserAccess  
);
```

Change the access rights associated with the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies one or more user access rights. For a list of user access rights that can be granted to the client, see [User Access Constants](#).

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero. This function can only be used with authenticated clients. If the client session has not been authenticated, the return value will be zero.

Remarks

The **FtpSetClientAccess** function can change the access rights for an authenticated client session. This function can only be used after the **FtpAuthenticateClient** function has been used to grant the initial set of access rights to the client. The **FtpEnableClientAccess** function can be used to grant or revoke a specific permission for the client session.

The *dwUserAccess* parameter has a value of FTP_ACCESS_DEFAULT, then default permissions will be granted to the client session. A normal client cannot be changed to a restricted or anonymous client using this function. If the FTP_ACCESS_RESTRICTED or FTP_ACCESS_ANONYMOUS access flags are specified, this function will fail.

This function cannot be used to change the access rights for a restricted or anonymous user. Those rights are granted when the client session is authenticated and will persist until the client disconnects from the server. This restriction is designed to prevent the inadvertent granting of rights to an untrusted client that could compromise the security of the server.

Example

```
DWORD dwUserAccess = 0;  
  
// Allow the client to execute programs using SITE EXEC  
if (FtpGetClientAccess(hServer, nClientId, &dwUserAccess))  
{  
    if (!(dwUserAccess & FTP_ACCESS_ANONYMOUS))  
        FtpSetClientAccess(hServer, nClientId, dwUserAccess |  
FTP_ACCESS_EXECUTE);  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[FtpAuthenticateClient](#), [FtpEnableClientAccess](#), [FtpGetClientAccess](#)

FtpSetClientFileType Function

```
BOOL WINAPI FtpSetClientFileType(  
    HSERVER hServer,  
    UINT nClientId,  
    UINT nFileType  
);
```

Change the current file type used for transfers by the specified client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

nFileType

Specifies the type of file that will be uploaded or downloaded. This parameter determines whether subsequent file transfers require any data conversion and may be one of the following values.

Value	Description
FILE_TYPE_ASCII (1)	The file is a text file using the ASCII character set. For those clients which use a different end-of-line character sequence, the text file has been converted to the local format which uses the carriage return (CR) and linefeed (LF) characters.
FILE_TYPE_IMAGE (3)	The file is a binary file and no data conversion of any type has been performed on the file. This is the default file type for most data files and executable programs. If the client specified this file type when appending to a text file, the file will contain the end-of-line sequences used by its native operating system.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **FtpSetClientFileType** function will change the default file type that is used for subsequent transfers by the client. If the file type is set to FILE_TYPE_ASCII then the server will automatically convert any end-of-line character sequences to match the format used by the local system. For example, if the client is connecting from a UNIX based system, the server will convert a single linefeed character to a carriage return (CR) and linefeed (LF) sequence. If the file type is set to FILE_TYPE_IMAGE, then no conversion is performed.

This function can be used to override the file type specified by filtering the TYPE command issued by the client. For example, it could be used to force file transfers to use a specific type based on the file extension, regardless of the type specified by the client. To determine the current file type set by the client, use the **FtpGetClientFileType** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

See Also

[FtpGetClientFileType](#)

FtpSetClientIdentity Function

```
BOOL WINAPI FtpSetClientIdentity(  
    HSERVER hServer,  
    UINT nClientId,  
    LPCTSTR lpszIdentity  
);
```

Change the identity of the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszIdentity

A pointer to a string that identifies the client. If this parameter is NULL or specifies an empty string, the current identity for the client is cleared. The maximum length of the identity string is 64 characters, including the terminating null character.

Return Value

If the function succeeds, the return value is non-zero. If the server handle or client ID is invalid, the function will return a value of zero.

Remarks

The **FtpSetClientIdentity** function associates a string value with the client that can be used to identify the session. The identity string does not have any standard format and is used for informational purposes only. Typically it is used to identify the client application that was used to establish the connection. Changing the client identity has no effect on the operation of the server. To obtain the identity string currently associated with the client, use the **FtpGetClientIdentity** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetClientIdentity](#), [FtpGetServerIdentity](#), [FtpSetServerIdentity](#)

FtpSetClientIdleTime Function

```
UINT WINAPI FtpSetClientIdleTime(  
    HSERVER hServer,  
    UINT nClientId,  
    UINT nTimeout  
);
```

Change the idle timeout period for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

nTimeout

An unsigned integer value that specifies the number of seconds that the client may remain idle. If this value is zero, the default idle timeout period for the server will be used.

Return Value

If the function succeeds, the return value is the previous client idle timeout period in seconds. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **FtpSetClientIdleTime** function will change the number of seconds that the client may remain idle before being automatically disconnected by the server. The minimum timeout period for a client is 60 seconds, the maximum is 7200 seconds (2 hours). The idle time of a client session is based on the last time a command was issued to the server or when a data transfer completed.

If the value INFINITE is specified as the timeout period, the client activity timer will be refreshed, extending the idle timeout period for the session. This is typically done inside an event handler to prevent the client from being disconnected due to inactivity.

To obtain the current idle timeout period for a client, along with the amount of time the client has been idle, use the **FtpGetClientIdleTime** function.

This timeout period only affects authenticated clients. Unauthenticated clients use a different internal timer that limits the amount of time they can remain connected to the server and that value cannot be changed for individual client sessions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

See Also

[FtpGetClientIdleTime](#)

FtpSetCommandFile Function

```
BOOL WINAPI FtpSetCommandFile(  
    HSERVER hServer,  
    UINT nClientId,  
    LPCTSTR lpszFileName  
);
```

Change the name of the local file or directory that is the target of the current command.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A pointer to a string that specifies the new file name. This parameter may be NULL to specify that the original file or directory name should be used.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **FtpSetCommandFile** function is used by the application to change the target file or directory name for the current command from within an FTP_EVENT_COMMAND event handler. This can be used to effectively redirect the client to use a different file than the one that was actually requested. For example, if the client issues the RETR command to download a file from the server, this function can be used to redirect the command to use a different file name. To obtain the full path to the file or directory that is the target of the current command, use the **FtpGetCommandFile** function.

The *lpszFileName* parameter specifies the path to the new file or directory name. If the path is absolute, then it will be used as-is. If the path is relative, it will be relative to the current working directory for the client session. The full path to this file is not limited to the server root directory or its subdirectory, it can specify a file anywhere on the local system. If this parameter is a NULL pointer, or points to an empty string, then the server will revert to using the actual file or directory name specified by the command. This enables the application to effectively undo a previous call to this function to change the target file name.

Typically this function would be used to redirect a client to a file or directory that it may not normally have access to. Exercise caution when using this function to provide access to data that is stored outside of the server root directory. Incorrect use of this function could expose the server to security risks or cause unpredictable behavior by client applications.

This function should only be called within the context of the FTP_EVENT_COMMAND event, and only for those commands that perform an action on a file or directory. If the current command does not target a file or directory, this function will return zero and the last error code will be set to ST_ERROR_INVALID_COMMAND. To obtain the name of the current command issued by the client, use the **FtpGetCommandName** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetCommandFile](#), [FtpGetCommandLine](#), [FtpGetCommandName](#)

FtpSetServerAddress Function

```
BOOL WINAPI FtpSetServerAddress(  
    HSERVER hServer,  
    LPCTSTR lpszAddress  
);
```

Change the IP address that the server will use with passive data connections.

Parameters

hServer

The server handle.

lpszAddress

A pointer to a string that specifies the IP address that the server should use for passive mode data transfers. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If either the server handle or the IP address is invalid the function will return a value of zero.

Remarks

The **FtpSetServerAddress** function changes the IP address that the server will use when a client transfers a file using a passive mode data connection. In passive mode, the server will create a second passive (listening) socket that will accept an incoming connection from the client. The server sends the IP address and port number allocated for that socket to the client, and the client establishes the data connection to the server using that address. The IP address that the server sends to the client is normally the same as the IP address that the client used to establish the control connection, however if the server is located behind a router that performs Network Address Translation (NAT), the IP address reported to the client may not be usable.

This function enables your application to set the external IP address for the server to a specific value, rather than the server attempting to automatically discover its own external address. If you wish to set the external address for the server manually, call the **FtpServerStart** function without the FTP_SERVER_EXTERNAL option and then call this function to set the external IP address to the desired value.

This function will not change the IP address the server is using to listen for client connections. The only way to change the listening IP address is to stop and restart the server using the new address. This function only changes the IP address that is reported to clients when a passive data connection is used. Incorrect use of this function can prevent the client from establishing a data connection to the server. The address must be in the same address family as the local address that the server was started with. For example, if the server was started using an IPv4 address, the IP address passed to this function cannot be an IPv6 address.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

FtpSetServerError Function

```
VOID WINAPI FtpSetServerError(  
    HSERVER hServer,  
    DWORD dwError  
);
```

Set the last error code for the specified server session.

Parameters

hServer

The server handle.

dwError

An unsigned integer that specifies an error code.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each server session. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SERVER or FTP_ERROR.

If the *hServer* parameter is specified as INVALID_SERVER, this function will set the last error code for the current thread, but will not change the error code associated with any server session. This should only be done if the application does not have access to a valid server handle.

If the *dwError* parameter is specified with a value of zero, this effectively clears the error code for the last function that failed. Those functions which clear the last error code when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **FtpGetServerError** function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

See Also

[FtpGetCommandResult](#), [FtpGetServerError](#), [FtpSendResponse](#)

FtpSetServerIdentity Function

```
BOOL WINAPI FtpSetServerIdentity(  
    HSERVER hServer,  
    LPCTSTR lpszIdentity  
);
```

Change the identity of the specified server.

Parameters

hServer

The server handle.

lpszIdentity

A pointer to a string that identifies the server. If this parameter is NULL or specifies an empty string, the current identity for the server is reset to a default value. The maximum length of the identity string is 64 characters, including the terminating null character.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid, the function will return a value of zero.

Remarks

The **FtpSetClientIdentity** function changes a string value used by the server to identify itself to clients. The identity string does not have any standard format and is used for informational purposes only. Typically it consists of the application name and a version number. Changing the server identity has no effect on the operation of the server. To obtain the identity string currently associated with the server, use the **FtpGetServerIdentity** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetClientIdentity](#), [FtpSetClientIdentity](#), [FtpGetServerIdentity](#)

FtpSetServerLogFile Function

```
BOOL WINAPI FtpSetServerLogFile(  
    HSERVER hServer,  
    UINT nLogFormat,  
    UINT nLogLevel,  
    LPCTSTR lpszFileName  
);
```

Change the current log format, level of detail and file name.

Parameters

hServer

The server handle.

nLogFormat

An integer value that specifies the format used when creating or updating the server log file. The following formats are supported:

Constant	Description
FTP_LOGFILE_NONE (0)	This value specifies that the server should not create or update a log file.
FTP_LOGFILE_COMMON (1)	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about file transfers.
FTP_LOGFILE_EXTENDED (2)	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session.

nLogLevel

An integer value that specifies the level of detail that should be generated in the log file. The minimum value is 1 and the maximum value is 10. If this parameter is zero, it is the same as specifying a log file format of FTP_LOGFILE_NONE and will disable logging by the server.

lpszFileName

A pointer to a string that specifies the name of the log file that should be created or appended to. If the server was configured with logging enabled and this parameter is NULL or an empty string, the current log file name will not be changed. If the log file does not exist, it will be created. If it does exist, the contents of the log file will be appended to.

Return Value

If the function succeeds, the return value is non-zero. If the server handle does not specify a valid server, the function will return zero.

Remarks

The **FtpSetServerLogFile** function can be used to change the current log file name, the format of the log file or the level of detail recorded in the log file. In some situations it may be desirable to delete the current log file contents when changing the format or ensure that a new log file is created. To do this, combine the *nLogFormat* parameter with the constant FTP_LOGFILE_DELETE.

The higher the value of the *nLogLevel* parameter, the greater the level of detail that is recorded by the server. A log level of 1 instructs the server to only record file transfers, while a level of 10 instructs the server to record all commands processed by the server. Because a higher level of logging detail can negatively impact the performance of the server, it is recommended that you do not exceed a level of 5 for most applications. A log level of 10 should only be used for debugging purposes.

Example

```
UINT nLogFormat = FTP_LOGFILE_NONE;
UINT nLogLevel = 0;
UINT nNewLevel = 5;
BOOL bChanged = FALSE;

// Change the level of detail for the current log file if logging
// has been enabled and the current level is a lower value

if (FtpGetServerLogFile(hServer, &nLogFormat, &nLogLevel, NULL, 0))
{
    if (nLogFormat != FTP_LOGFILE_NONE && nLogLevel < nNewLevel)
        bChanged = FtpSetServerLogFile(hServer, nLogFormat, nNewLevel, NULL);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetServerLogFile](#), [FtpRenameServerLogFile](#)

FtpSetServerName Function

```
BOOL WINAPI FtpSetServerName(  
    HSERVER hServer,  
    UINT nClientId,  
    LPCTSTR lpszHostName  
);
```

Change the host name assigned to the specified server or client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszHostName

A pointer to a string that specifies the new host name assigned to the server or client session. If this value is NULL or points to an empty string, the current host name will be changed to use the default host name.

Return Value

If the function succeeds, the return value is non-zero. If either the server handle or the client ID is invalid, or the buffer is not large enough to store the complete hostname, the function will return a value of zero.

Remarks

This function will change the host name assigned to the specified client session. If the *nClientId* parameter has a value of zero, the function will change default host name that was assigned to the server as part of the server configuration. If the *nClientId* parameter specifies a valid client session and the *lpszHostName* parameter is NULL, the host name associated with the client session will be changed to the current host name assigned to the server.

When a client connects to the server, it can specify the host name that it used to establish the connection by sending the HOST command. This is typically used with virtual hosting, where one server may accept client connections for multiple domains. The **FtpGetServerName** function will return the host name specified by the client, and **FtpSetServerName** can be used by the application to either explicitly assign a different host name to the client session, or override the host name provided by the client.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csftsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetServerAddress](#), [FtpGetServerName](#)

FtpSetServerPriority Function

```
INT WINAPI FtpSetServerPriority(  
    HSERVER hServer,  
    INT nPriority  
);
```

Change the priority assigned to the specified server.

Parameters

hServer

The server handle.

nPriority

An integer value which specifies the new priority for the server. It may be one of the following values:

Constant	Description
FTP_PRIORITY_BACKGROUND (0)	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
FTP_PRIORITY_LOW (1)	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
FTP_PRIORITY_NORMAL (2)	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
FTP_PRIORITY_HIGH (3)	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
FTP_PRIORITY_CRITICAL (4)	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Return Value

If the function succeeds, the return value is the previous priority assigned to the server. If the function fails, the return value is FTP_PRIORITY_INVALID. To get extended error information, call **FtpGetServerError**.

Remarks

The **FtpSetServerPriority** function can be used to change the current priority assigned to the specified server. Client connections that are accepted after this function is called will inherit the new priority as their default priority. Previously existing client connections will not be affected by this function.

Higher priority values increase the thread priority and processor utilization for each client session. You should only change the server priority if you understand the impact it will have on the system and have thoroughly tested your application. Configuring the server to run with a higher priority can have a negative effect on the performance of other programs running on the system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cstools10.h`.

Import Library: `csftsv10.lib`

See Also

[FtpGetServerPriority](#), [FtpServerStart](#)

FtpSetServerStackSize Function

```
BOOL WINAPI FtpSetServerStackSize(  
    SOCKET hServer,  
    DWORD dwStackSize  
);
```

Change the initial size of the stack allocated for threads created by the server.

Parameters

hServer

Handle to the server socket.

dwStackSize

The amount of memory that will be committed to the stack for each thread created by the server. If this value is zero, a default stack size will be used.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **FtpGetServerError**.

Remarks

The **FtpSetServerStackSize** function changes the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes. Increasing or decreasing the stack size will only affect new threads that are created by the server, it will not affect those threads that have already been created to manage active client sessions. It is recommended that most applications use the default stack size.

You should not change the stack size unless you understand the impact that it will have on your system and have thoroughly tested your application. Increasing the initial commit size of the stack will remove pages from the total system commit limit, and every page of memory that is reserved for stack cannot be used for any other purpose.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `csftools10.h`

Import Library: `csftsv10.lib`

See Also

[FtpGetServerMemoryUsage](#), [FtpGetServerStackSize](#), [FtpServerStart](#)

File Transfer Protocol Server Data Structures

- FTPCLIENTCREDENTIALS
- FTPSERVERCONFIG
- FTPSERVERTRANSFER
- INITDATA
- SECURITYCREDENTIALS

FTPCLIENTCREDENTIALS Structure

The **FTPCLIENTCREDENTIALS** structure defines the credentials used to authenticate a specific user.

```
typedef struct _FTPCLIENTCREDENTIALS
{
    DWORD dwSize;
    DWORD dwFlags;
    TCHAR szHostName[FTP_MAXHOSTNAME];
    TCHAR szUserName[FTP_MAXUSERNAME];
    TCHAR szPassword[FTP_MAXPASSWORD];
} FTPCLIENTCREDENTIALS, *LPFTPCLIENTCREDENTIALS;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwFlags

An unsigned integer value reserved for future use. This member will always be initialized to a value of zero.

szHostName

A pointer to a string that specifies the server host name.

szUserName

A pointer to a string that specifies the user name.

szPassword

A pointer to a string that specifies the user password.

Remarks

When an instance of this structure is passed to the **FtpGetClientCredentials** function, this member must be initialized to the size of the structure and all other members must be initialized with a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetClientCredentials](#)

FTPSERVERCONFIG Structure

The **FTPSERVERCONFIG** structure provides the configuration information used to create an instance of an FTP server.

```
typedef struct _FTPSERVERCONFIG
{
    DWORD    dwSize;
    UINT     nMaxClients;
    UINT     nMaxClientsPerAddress;
    UINT     nMaxGuests;
    UINT     nLogFormat;
    UINT     nLogLevel;
    UINT     nMinPort;
    UINT     nMaxPort;
    UINT     nAuthFail;
    UINT     nAuthTime;
    UINT     nIdleTime;
    UINT     nExecTime;
    LPCTSTR  lpszIdentity;
    LPCTSTR  lpszHostUuid;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszRootPath;
    LPCTSTR  lpszTempPath;
    LPCTSTR  lpszLogFile;
} FTPSERVERCONFIG, *LPFTPSERVERCONFIG;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

nMaxClients

An integer value that specifies the maximum number of active client connections that will be accepted by the server. If the maximum number of clients is reached, any further connections are rejected by the server until one or more clients close their connection to the server or are disconnected. A value of zero specifies the default configuration value of 100 connections should be used.

nMaxClientsPerAddress

An integer value that specifies the maximum number of active client connections per IP address that will be accepted by the server. If the maximum number of clients from the same IP address is reached, any further connections are rejected until one of the clients closes its connection to the server. A value of zero specifies the default configuration value of 4 connections per address should be used.

nMaxGuests

An integer value that specifies the maximum number of anonymous client connections. If the server does not permit anonymous connections, this value is ignored. It is recommended that this value always be less than the value of the *nMaxClients* member. A value of zero specifies that the default configuration limit of 20 anonymous connections should be used.

nLogFormat

An integer value that specifies the format used when creating or updating the server log file. The following formats are supported:

--	--

Constant	Description
FTP_LOGFILE_NONE (0)	This value specifies that the server should not create or update a log file.
FTP_LOGFILE_COMMON (1)	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about file transfers.
FTP_LOGFILE_EXTENDED (2)	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session.

nLogLevel

An integer value that specifies the level of detail that should be generated in the log file. The minimum value is 1 and the maximum value is 10. If the ***nLogFormat*** member specifies a valid log file format and this value is zero, a default level of detail will be selected based on the format. The common log file format generally contains less information by default, only logging the data transfers between the client and server. The W3C extended log file format defaults to a higher level of detail that includes additional information about the client session. The higher the level of detail, the larger the log file will be.

nMinPort

An integer value that specifies the minimum range of port numbers that will be used with passive data connections. A value of zero specifies that the default value of 30000 should be used. The minimum value of this member is 5000 and the maximum value is 65535. If the value is non-zero, it must be less than the value of the ***nMaxPort*** structure member.

nMaxPort

An integer value that specifies the maximum range of port numbers that will be used with passive data connections. A value of zero specifies the default value of 65535 should be used. The minimum value of this member is 5000 and the maximum value is 65535. If the value is non-zero, it must be greater than the value of the ***nMinPort*** structure member.

nAuthFail

An integer value that specifies the maximum number of user authentication attempts that are permitted until the server terminates the client connection. A value of zero specifies that the default configuration limit of 3 authentication attempts per login should be allowed. The maximum number of authentication attempts is 10.

nAuthTime

An integer value that specifies the maximum number of user authentication attempts that are permitted until the server terminates the client connection. A value of zero specifies the default value of 60 seconds. If the value is non-zero, the minimum value is 20 seconds and the maximum value is 300 seconds (5 minutes). This value is used to ensure that a client has successfully authenticated itself within a limited period of time. This prevents a potential denial-of-service attack against the server where clients establish connections and hold them open without authentication. In conjunction with the ***nAuthFail*** member, this also limits the ability of a client to attempt to probe the server for valid username and password combinations.

nIdleTime

An integer value that specifies the maximum number of seconds that a client session may be

idle before the server closes the control connection to the client. A value of zero specifies the default value of 900 seconds (15 minutes). If the value is non-zero, the minimum value is 60 seconds and the maximum value is 7200 seconds (2 hours). This value is used to initialize the default idle timeout period for each client session. A client may request that the server change the idle timeout period for its session by sending the SITE IDLE command. The server determines if a client is idle based on the time the last command was issued and whether or not a file transfer is in progress.

nExecTime

An unsigned integer value that specifies the maximum number of seconds that an external program is permitted to run on the server. External programs are registered using the **FtpRegisterProgram** function, and are executed by the client sending the SITE EXEC command to the server. If this value is zero, the default timeout period of 5 seconds will be used. The minimum execution time is 1 second and the maximum time limit is 30 seconds.

lpszIdentity

A pointer to a string that identifies the server. It is used for informational purposes only and has no effect on the operation of the server. If this member is not initialized to a value, then a default identity will be used. The server identity is provided when a client establishes the initial connection and when the client sends the STAT or CSID commands. If this member is defined, it is recommended that you only use printable ASCII characters.

lpszHostUuid

A pointer to a string that specifies a Universally Unique Identifier (UUID) that is used to uniquely identify the server. This value can be used when storing information about the server, and should be generated using a utility such as **uuidgen** which is included with Visual Studio. This structure member may be initialized to an empty string, in which case a temporary UUID will be randomly generated.

lpszHostName

A pointer to a string that specifies the fully-qualified host name for the server. If this structure member is initialized with an empty string, the local host name assigned to the Windows system will be used. This value does not need to correspond to the actual host name associated with the server's IP address. This value is used for informational purposes only and has no effect on the operation of the server.

lpszRootPath

A pointer to a string that specifies the path to the root directory for the server. If this structure member is initialized with an empty string, the current working directory for the process will be used as the root directory. It is recommended that you provide a full path to an existing directory and do not specify the root of a local or network drive. If the path does not exist at the time the server is started, it will be automatically created.

lpszTempPath

A pointer to a string that specifies the path to the temporary directory for the server. If this structure member is initialized with an empty string, the current temporary directory for the process will be used. The temporary directory cannot be the same as the root directory, and it is strongly recommended that you do not specify the root of a local or network drive. If the path does not exist at the time the server is started, it will be automatically created.

lpszLogFile

A pointer to a string that specifies the name of the server log file to create, if a logging format has been specified. If logging is enabled and this member is an empty string, then a default log file name will be created. If the file name does not include a path, then the file is created in the

server log directory. If the file name includes a path, the log file will be created using that specific name. If the server is in multi-user mode, then the default location for log files will be the Logs subdirectory in the server root directory. If the server is not in multi-user mode, the default location for log files will be the temporary directory for the server process.

Remarks

When an instance of this structure is passed to the **FtpServerStart** function, the *dwSize* member must be initialized to the size of the structure, otherwise the function will fail with an error indicating that the configuration is invalid.

The *nMaxClients* member limits the total number of client connections and the *nMaxGuests* member limits the total number of guest connections. The server will notify the client if either number of connections has been exceeded and will automatically close the connection. However, one difference is that a new client that exceeds the maximum number of connections will not generate any events; a client that exceeds the maximum number of guest connections will generate events. The reason for the difference is that the check for the number of active clients is made immediately after the client connects to the server, prior to session context being created for that client. The check for the number of anonymous clients is made later, after the client has sent the USER and PASS commands to the server. For more control over how the server accepts client connections, use the **FtpServerThrottle** function.

If the server is running on a system that is behind a firewall that restricts the range of port numbers for inbound connections, the *nMinPort* and *nMaxPort* members can be used to limit the ports which the server will use to listen for passive data connections from clients. By default, the server will use data port numbers that range from 30000 to 65535, which is typical for many firewall configurations. If you provide your own port range, the minimum number of ports available must be at least 1000. Specifying a port range that is too small will cause the **FtpServerStart** function to fail with an error indicating that the server configuration is invalid.

When specifying the root directory for the server configuration, it is important to consider that a client's access to files on the system will be limited to these directories and any subdirectories. If a root directory is not specified, then one will be created using the current working directory for the process. The root directory path may include environment variables surrounded by % symbols and these will be expanded.

If you have configured the server to permit clients to upload files, you must ensure that your application has permission to create files in the directory that you specify. A recommended location for the server root directory would be a subdirectory of the %ALLUSERSPROFILE% directory. Using the environment variable ensures that your server will work correctly on different versions of Windows. If the root directory does not exist at the time that the server is started, it will be created.

For servers that are publicly accessible, or where you want files to be accessible across multiple server sessions, you should always populate the *szHostUuid* member with a valid UUID string, and the *szDirectory* member should specify an absolute path to an existing directory.

If the FTP_SERVER_MULTUSER option is specified, the server will start in multi-user mode, with each user assigned to their own home directory based on their username. This option will cause the server to create a directory structure in the root directory that includes three subdirectories: Public, Users and Logs. The Public directory is the home directory for anonymous users, and they are isolated to that directory. The Users directory is where the user home directories are created. The Logs directory is where log files are created by default. If this option is not specified, then no subdirectories are created and all users will share the server root directory by default.

If the `FTP_SERVER_LOCALUSER` option is specified, the server will never attempt to authenticate the local Administrator or Guest accounts. This is done to prevent clients from attempting to probe for the administrator account password. However, the application can override this behavior by implementing a handler for the `FTP_CLIENT_LOGIN` event.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpServerStart](#), [FtpServerStop](#), [FtpServerThrottle](#)

FTPSEVERTRANSFER Structure

The **FTPSEVERTRANSFER** structure provides information about the last file transfer performed by a client.

```
typedef struct _FTPSEVERTRANSFER
{
    DWORD          dwSize;
    DWORD          dwReserved;
    DWORD          dwFileAccess;
    DWORD          dwTimeElapsed;
    ULARGE_INTEGER uiBytesCopied;
    TCHAR          szFileName[MAX_PATH];
} FTPSEVERTRANSFER, *LPFTPSEVERTRANSFER;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwReserved

An unsigned integer value that is reserved for future use. This value will always be zero.

dwFileAccess

An unsigned integer value that specifies the how the local file was accessed. It can be one of the following values:

Constant	Description
FTP_FILE_READ (0)	The file was opened for reading. This mode indicates that the client issued the RETR command to download the contents of a file from the server to the client system. The <i>szFileName</i> member specifies the name of the local file on the server that was downloaded by the client.
FTP_FILE_WRITE (1)	The file was opened for writing. This mode indicates that the client issued the STOR or STOU command to upload the contents of a file from the client system to server. The <i>szFileName</i> member specifies the name of the local file on the server that was created by the client. If a file already existed with the name name, it was replaced.
FTP_FILE_APPEND (2)	The file was opened for writing. This mode indicates the client issued the APPE command to upload the contents of a file from the client system and append the data to a file on the server. If the file did not exist, then it was created. The <i>szFileName</i> member specifies the name of the local file that was appended to or created by the client.

dwTimeElapsed

The amount of time that it took for the file transfer to complete in milliseconds. This value is limited to the resolution of the system timer, which is typically in the range of 10 to 16 milliseconds. This value may be zero if the transfer occurred over a local network or on the same host using a loopback address.

uiBytesCopied

A 64-bit integer value that specifies the total number of bytes copied during the file transfer. This value is represented by a `ULARGE_INTEGER` union which provides support for those programming languages that do not have intrinsic support for 64-bit integers. For more information, refer to the Windows SDK documentation. The application should not make the assumption that this is the actual size of the file. If the client specified a restart offset using the REST command, this value would only represent the number of bytes transferred from that byte offset, not the total file size.

szFileName

A pointer to a string value that will contain the full path to the local file that was transferred. The ***dwFileAccess*** member determines whether the file name represents a file that was downloaded by the client, or uploaded from the client and stored on the server.

Remarks

When an instance of this structure is passed to the **FtpGetServerTransferInfo** function, the ***dwSize*** member must be initialized to the size of the structure, otherwise the function will fail with an error indicating that the parameter is invalid. All other members should be initialized to a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[FtpGetServerTransferInfo](#)

INITDATA Structure

This structure contains information about the SocketTools library when the initialization function returns.

```
typedef struct _INITDATA
{
    DWORD        dwSize;
    DWORD        dwVersionMajor;
    DWORD        dwVersionMinor;
    DWORD        dwVersionBuild;
    DWORD        dwOptions;
    DWORD_PTR    dwReserved1;
    DWORD_PTR    dwReserved2;
    TCHAR        szDescription[128];
} INITDATA, *LPINITDATA;
```

Members

dwSize

Size of this structure. This structure member must be set to the size of the structure prior to calling the initialization function. Failure to do so will cause the function to fail.

dwVersionMajor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the major version number for the library.

dwVersionMinor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the minor version number for the library.

dwVersionBuild

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the build number for the library.

dwOptions

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved1

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved2

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

szDescription

A null terminated string which describes the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

SECURITYCREDENTIALS Structure

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on

	what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that

will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A pointer to a null terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A pointer to a null terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A pointer to a null terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A pointer to a null terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
------------	-------------

CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
----	---

MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
----	--

ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.
------	--

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

A pointer to a null terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used for SSH connections

and should always be NULL when establishing a secure connection using SSL or TLS.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* member should be set to CREDENTIAL_STORE_FILENAME and the *lpzCertStore* member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS#12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the *lpzPassword* member or the library will be unable to access the certificate.

Note that the *lpzUserName* and *lpzPassword* members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

The TLS 1.1 and TLS 1.2 protocols are only supported on Windows 7, Windows Server 2008 R2 and later versions of the platform. If these options are specified and the application is running on Windows XP or Windows Vista, the protocol version will be downgraded to TLS 1.0 for backwards compatibility with those platforms.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

Hypertext Transfer Protocol Client Library

Transfer files between the local system and a web server, execute scripts and perform remote file management functions.

Reference

- [Functions](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

File Name	CSHTPV10.DLL
Version	10.0.1468.2518
LibID	FAA72D93-4063-474E-97DE-25BA304E1098
Import Library	CSHTPV10.LIB
Dependencies	None
Standards	RFC 1945, RFC 2616

Overview

The Hypertext Transfer Protocol (HTTP) is a lightweight, stateless application protocol that is used to access resources on web servers, as well as send data to those servers for processing. The library provides direct, low-level access to the server and the commands that are used to retrieve resources (i.e.: documents, images, etc.). The library also provides a simple interface for downloading resources to the local host, similar to how the HTTP library can be used to download files.

In a typical session, the library is used to establish a connection, send a request (to download a resource, post data for processing, etc.), read the data returned by the server and then disconnect. It is the responsibility of the client to process the data returned by the server, depending on the type of resource that was requested.

This library supports secure connections using the standard SSL and TLS protocols.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Hypertext Transfer Protocol Functions

Function	Description
HttpAddFormField	Add the form field and its value to the specified form
HttpAddFormFile	Add the contents of the file to the specified form
HttpAsyncConnect	Establish an asynchronous connection to the specified server
HttpAsyncGetData	Copy the contents of a resource on the server to a local buffer
HttpAsyncGetFile	Copy a file from the server to the local system
HttpAsyncGetFileEx	Copy a file from the server to the local system, use with files larger than 4GB
HttpAsyncPostData	Post data from a local buffer to a script executed on the server
HttpAsyncPostJson	Post JSON formatted data to a script executed on the server
HttpAsyncPostXml	Post XML formatted data to a script executed on the server
HttpAsyncProxyConnect	Establish an asynchronous connection to a proxy server
HttpAsyncPutData	Create a file on the server using the contents of a local buffer
HttpAsyncPutFile	Copy a file from the local system to the server
HttpAsyncPutFileEx	Copy a file from the local system to the server, use with files larger than 4GB
HttpAsyncSubmitForm	Submit the specified form to the server for processing
HttpAttachThread	Attach the specified client handle to another thread
HttpAuthenticate	Specify authentication information for restricted resources
HttpCancel	Cancel the current blocking operation
HttpClearForm	Remove all defined fields from the specified form
HttpCloseFile	Close the file opened on the server
HttpCommand	Send a command to the server
HttpConnect	Connect to the specified server
HttpConnectUrl	Establish a client connection using the specified URL
HttpCreateFile	Create or replace a file on the server
HttpCreateForm	Create a new form and return the form handle
HttpCreateSecurityCredentials	Create a new security credentials structure
HttpDeleteFile	Remove a file from the server
HttpDeleteFormField	Delete the form field and its value from the specified form
HttpDeleteHeaders	Delete all of the response or request headers for the current session
HttpDeleteSecurityCredentials	Delete a previously created security credentials structure
HttpDestroyForm	Destroy the specified form and free the memory allocated for it

HttpDisableEvents	Disable asynchronous event notification
HttpDisableTrace	Disable logging of socket function calls to the trace log
HttpDisconnect	Disconnect from the current server
HttpDownloadFile	Download a file from the server to the local system
HttpEnableCompression	Enable or disable support for data compression
HttpEnableEvents	Enable asynchronous event notification
HttpEnableTrace	Enable logging of socket function calls to a file
HttpEnumTasks	Return a list of asynchronous tasks
HttpEventProc	Callback function that processes events generated by the client
HttpFreezeEvents	Suspend asynchronous event processing
HttpGetBearerToken	Return the current OAuth 2.0 bearer token for the client session
HttpGetCookie	Return information about the specified cookie
HttpGetData	Copy the specified resource to a local buffer
HttpGetEncodingType	Determines which content encoding option is enabled
HttpGetErrorString	Return a description for the specified error code
HttpGetFile	Copy a file from the server to the local system
HttpGetFileEx	Copy a file from the server to the local system, use with large files over 4GB
HttpGetFileSize	Return the size of a file on the server
HttpGetFileSizeEx	Return the size of a file on the server, supports large files over 4GB
HttpGetFileTime	Return the date and time a file on the server was last modified
HttpGetFirstCookie	Return the first cookie set by the server
HttpGetFirstHeader	Return the name and value of the first request or response header field
HttpGetFormProperties	Return the properties of the specified form
HttpGetLastError	Return the last error code
HttpGetNextCookie	Return the next cookie set by the server
HttpGetNextHeader	Return the name and value of the next request or response header field
HttpGetOption	Return the enabled/disabled state of a specified option
HttpGetPriority	Return the current priority for file transfers
HttpGetRequestHeader	Return the value of the specified request header field
HttpGetResponseHeader	Return the value of the specified response header field
HttpGetResultCode	Return the result code from the previous command
HttpGetResultString	Return the result string from the previous command
HttpGetSecurityInformation	Return security information about the current client connection
HttpGetStatus	Return the current client status

HttpGetTaskError	Return the last error code for the specified asynchronous task
HttpGetTaskId	Return the unique task identifier associated with the specified client session
HttpGetText	Download the contents of a text file or resource to a string buffer
HttpGetTimeout	Return the number of seconds until an operation times out
HttpGetTransferStatus	Return data transfer statistics
HttpInitialize	Initialize the library and validate the specified runtime license key
HttpIsBlocking	Determine if the client is blocked, waiting for information
HttpIsConnected	Determine if the client is connected to the server
HttpIsReadable	Determine if data can be read from the server
HttpIsWritable	Determine if data can be written to the server
HttpOpenFile	Open a file on the server for reading
HttpPatchData	Submits JSON or XML patch data to the server and returns the response
HttpPostData	Submit data to the server using the POST command and returns the response
HttpPostFile	Submit the contents of a local file to the server
HttpPostJson	Post JSON formatted data to the server and returns the response
HttpPostXml	Post XML formatted data to the server and returns the response
HttpProxyConnect	Establish a connection with the specified proxy server
HttpPutData	Create a file on the server using the contents of a local buffer
HttpPutDataEx	Submit data to the server using the PUT command and returns the response
HttpPutFile	Copy a file from the local system to the server
HttpPutFileEx	Copy a file from the local system to the server, use with large files over 4GB
HttpPutText	Create a text file on the server from the contents of a string buffer
HttpPutTextEx	Submit text to the server using the PUT command and returns the response
HttpRead	Read data from the server
HttpRegisterEvent	Register an event callback function
HttpSetBearerToken	Set the value of the OAuth 2.0 bearer token for the client session
HttpSetCookie	Set the value of the specified cookie
HttpSetEncodingType	Specifies the type of encoding to be applied to data submitted to the server
HttpSetFormProperties	Modify the properties of the specified form
HttpSetLastError	Set the last error code
HttpSetOption	Enable or disable the specified option

HttpSetPriority	Set the priority for file transfers
HttpSetRequestHeader	Set the value of a request header field
HttpSetTimeout	Set the number of seconds until an operation times out
HttpSubmitForm	Submit the specified form to the server for processing
HttpTaskAbort	Abort the specified asynchronous task
HttpTaskDone	Determine if an asynchronous task has completed
HttpTaskResume	Resume execution of an asynchronous task
HttpTaskSuspend	Suspend execution of an asynchronous task
HttpTaskWait	Wait for an asynchronous task to complete
HttpUninitialize	Terminate use of the library by the application
HttpUploadFile	Upload a file from the local system to the server
HttpValidateUrl	Check the contents of a string to ensure it represents a valid URL
HttpVerifyFile	Compare the size of a local file against a file stored on the server
HttpWrite	Write data to the server

HttpAddFormField Function

```
INT WINAPI HttpAddFormField(  
    HFORM hForm,  
    LPCTSTR lpszFieldName,  
    LPVOID lpFieldData,  
    DWORD cbFieldData,  
    DWORD dwReserved  
);
```

The **HttpAddFormField** function adds a new field to the specified form.

Parameters

hForm

Handle to the virtual form.

lpszFieldName

A pointer to a string which specifies the name of the field to add to the form. If this parameter is NULL or points to an empty string, then a default field name will be assigned.

lpFieldData

A pointer to the form field data. Typically this will either be a pointer to an array of bytes or a string which specifies the value for the form field. If no data is to be associated with the form field, then this argument may be NULL.

cbFieldData

An unsigned integer value which specifies the number of bytes of data in the form field. If this value is 0xFFFFFFFF (-1) then it is assumed that the *lpFieldData* parameter is a pointer to a null-terminated string. If *lpFieldData* is NULL, this value must be zero.

dwReserved

A reserved parameter. This value must be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpAddFormField** function is used to add a field and its associated value to a form created using the **HttpCreateForm** function. If the field name has already been added to the form, the previous value is deleted and replaced by the new value.

Example

```
HFORM hForm = INVALID_FORM;  
HGLOBAL hgb1Result = (HGLOBAL)NULL;  
DWORD cbResult = 0;  
INT nResult = 0;  
  
hForm = HttpCreateForm(_T("/login.php"), HTTP_METHOD_POST, HTTP_FORM_ENCODED);  
  
if (hForm == INVALID_FORM)  
    return;  
  
HttpAddFormField(hForm, _T("UserName"), lpszUserName, (DWORD)-1L, 0);  
HttpAddFormField(hForm, _T("Password"), lpszPassword, (DWORD)-1L, 0);
```

```
nResult = HttpSubmitForm(hClient, hForm, &hgblResult, &cbResult, 0);
HttpDestroyForm(hForm);

if (hgblResult != NULL)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblResult);

    // lpBuffer points to data returned by the server after the form
    // data was submitted

    GlobalUnlock(hgblResult);
    GlobalFree(hgblResult);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAddFormFile](#), [HttpClearForm](#), [HttpCreateForm](#), [HttpDeleteFormField](#), [HttpSubmitForm](#)

HttpAddFormFile Function

```
INT WINAPI HttpAddFormFile(  
    HFORM hForm,  
    LPCTSTR lpszFieldName,  
    LPCTSTR lpszFileName,  
    DWORD dwReserved  
);
```

The **HttpAddFormFile** function adds the contents of a file to the specified form.

Parameters

hForm

Handle to the virtual form.

lpszFieldName

A pointer to a string which specifies the name of the field to add to the form. If this parameter is NULL or points to an empty string, then a default field name will be assigned.

lpszFileName

A pointer to a string which specifies the name of the file. The contents of the file will be added to the form data that is submitted to the server.

dwReserved

A reserved parameter. This value must be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAddFormField](#), [HttpClearForm](#), [HttpCreateForm](#), [HttpDeleteFormField](#), [HttpPostFile](#), [HttpSubmitForm](#)

HttpAsyncConnect Function

```
HCLIENT WINAPI HttpAsyncConnect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    DWORD dwVersion,  
    LPSECURITYCREDENTIALS lpCredentials,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **HttpAsyncConnect** function is used to establish a connection with the server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **HttpConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to. This may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 80. For secure connections, the default port number is 443.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested. If the server does not support

	<p>the keep-alive option, the client will automatically reconnect when each resource is requested. Although it will not provide any performance benefits, this allows the option to be used with all servers.</p>
HTTP_OPTION_REDIRECT	<p>This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.</p>
HTTP_OPTION_PROXY	<p>This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed using the system Control Panel.</p>
HTTP_OPTION_ERRORDATA	<p>This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.</p>
HTTP_OPTION_TUNNEL	<p>This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.</p>
HTTP_OPTION_TRUSTEDSITE	<p>This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.</p>
HTTP_OPTION_SECURE	<p>This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.</p>
HTTP_OPTION_SECURE_FALLBACK	<p>This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and</p>

	cipher suites that use RC4, MD5 and SHA1.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
HTTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

dwVersion

The requested protocol version used when sending requests to the server. The high word should specify the major version, and the low word should specify the minor version number. The HTTPVERSION macro can be used to create version value.

lpCredentials

Pointer to credentials structure [SECURITYCREDENTIALS](#). This parameter is only used if the HTTP_OPTION_SECURE option is specified for the connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields *dwSize*, *lpzCertStore*, and *lpzCertName* must be defined, while other fields may be left undefined. Set *dwSize* to the size of the **SECURITYCREDENTIALS** structure.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **HttpGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the

following event identifiers may be sent:

Constant	Description
HTTP_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
HTTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
HTTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
HTTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
HTTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
HTTP_EVENT_PROGRESS	The client is in the process of sending or receiving data from the server. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.
HTTP_EVENT_REDIRECT	This event is generated when a the server indicates that the requested resource has been moved to a new location. The new resource location may be on the same server, or it may be located on another server. Check the value of the Location header field to determine where the resource has been moved to.

To cancel asynchronous notification and return the client to a blocking mode, use the **HttpDisableEvents** function.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the

HttpAttachThread function.

Specifying the HTTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAsyncProxyConnect](#), [HttpConnect](#), [HttpDisconnect](#), [HttpInitialize](#), [HttpProxyConnect](#)

HttpAsyncGetData Function

```
UINT WINAPI HttpAsyncGetData(  
    HCLIENT hClient,  
    LPCTSTR lpszResource,  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwOptions,  
    HTTPEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

Copies the contents of a resource on the server to the specified buffer.

Parameters

hClient

Handle to the client session.

lpszResource

A pointer to a string that specifies the resource on the server that will be transferred to the local system.

lpvBuffer

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvBuffer* parameter. If the *lpvBuffer* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the file that was downloaded.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_TRANSFER_DEFAULT	The default transfer mode. The resource data is copied to the local system exactly as it is stored on the server.
HTTP_TRANSFER_CONVERT	If the resource being downloaded from the server is textual, the data is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences.
HTTP_TRANSFER_COMPRESS	This option informs the server that the client is willing to accept compressed data. If the server supports compression for the specified resource, then the data will be automatically expanded before being returned to the caller. This option is selected by default if compression has been enabled using the

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **HttpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **HttpAsyncGetData** function is used to download the contents of a server resource into a local buffer. This function is similar to the **HttpGetData** function, however it retrieves the contents of the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being copied to the local system.

Because this function works asynchronously, it is important that the memory allocated for the buffer is not released before the asynchronous task completes. If you provide a buffer that is allocated on the stack, such as with the example listed below, then you must ensure that your code does not return from the function while the data is being downloaded. In the example, this is achieved by calling the **HttpTaskWait** function. You can also perform other operation and poll the status of the task by calling the **HttpTaskDone** function. If you wish to return from the calling function immediately, then you must dynamically allocate memory for the *lpvBuffer* and *lpdwLength* parameters on the heap and free that memory after the task has completed and the data is no longer needed.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the HTTP_EVENT_CONNECT event after the connection has been established, and the HTTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the data transfer is about to begin, and immediately before the worker thread is terminated. The worker thread creates a secondary connection to the server with its own session handle. This ensures that the asynchronous operation will not interfere with the current client session. Your application can interact with this background worker thread using the client handle that is passed to the event handler.

The *lpvBuffer* parameter may be specified in one of two ways, depending on the needs of the application. It can either be a pre-allocated buffer large enough to store the contents of the file or it can specify the address of a global memory handle that will contain the data. If it points to a pre-allocated buffer, the *lpdwLength* parameter must be initialized to the maximum number of bytes that can be copied into the buffer. If specifies the address of a global memory handle, then *lpdwLength* must be initialized to a value of zero. See the example code below.

Example

```
HGLOBAL hgb1Buffer = NULL;
```

```

DWORD dwLength = 0;
UINT nTaskId;

// Copy the data into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgb1Buffer parameter

nTaskId = HttpAsyncGetData(hClient,
                           lpszResource,
                           &hgb1Buffer,
                           &dwLength,
                           HTTP_TRANSFER_DEFAULT,
                           NULL, 0);

if (nTaskId != 0)
{
    DWORD dwError = NO_ERROR;
    DWORD dwElapsed = 0;

    // Wait for the transfer to complete
    HttpTaskWait(nTaskId, INFINITE, &dwElapsed, &dwError);

    // Lock the global memory handle, returning a pointer to the
    // contents of the file data
    lpBuffer = (LPBYTE)GlobalLock(hgb1Buffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgb1Buffer);
    GlobalFree(hgb1Buffer);
}

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAsyncGetFile](#), [HttpAsyncPutData](#), [HttpAsyncPutFile](#), [HttpEventProc](#), [HttpGetFile](#), [HttpTaskWait](#)

HttpAsyncGetFile Function

```
UINT WINAPI HttpAsyncGetFile(  
    HCLIENT hClient,  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszRemoteFile,  
    DWORD dwOptions,  
    DWORD dwOffset,  
    HTTPEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

Downloads the specified file from the server to the local system.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A pointer to a string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A pointer to a string that specifies the file on the server that will be transferred to the local system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Constant	Description
HTTP_TRANSFER_DEFAULT	The default transfer mode. The resource data is copied to the local system exactly as it is stored on the server.
HTTP_TRANSFER_CONVERT	If the resource being downloaded from the server is textual, the data is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences.
HTTP_TRANSFER_COMPRESS	This option informs the server that the client is willing to accept compressed data. If the server supports compression for the specified resource, then the data will be automatically expanded before being returned to the caller. This option is selected by default if compression has been enabled using the HttpEnableCompression function. This option is ignored if the <i>dwOffset</i> parameter is non-zero.

dwOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero

is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **HttpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **HttpAsyncGetFile** function is used to download the contents of a remote file to a file on the local system. This function is similar to the **HttpGetFile** function, however it retrieves the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being transferred to the local system.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the HTTP_EVENT_CONNECT event after the connection has been established, and the HTTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated. During the file transfer, the callback function will be invoked periodically with the HTTP_EVENT_PROGRESS event. The client session handle is passed to the event handler, allowing you to call functions such as **HttpGetTransferStatus** to determine the amount of data that has been copied.

To determine when the transfer has completed without implementing an event handler, periodically call the **HttpTaskDone** function. If you wish to block the current thread and wait for the transfer to complete, call the **HttpTaskWait** function. To stop a background file transfer that is in progress, call the **HttpTaskAbort** function. This will signal the background worker thread to cancel the transfer and terminate the session.

This function can be called multiple times to download multiple files in the background; however, most servers limit the number of simultaneous connections that can originate from a single IP address. It is recommended that you only perform two simultaneous background transfers from the same server at any one time. The application should not make any assumptions about the order in which multiple background transfers may complete or how they are sequenced. For example, it should never be assumed that a background task with a lower task ID will complete before a task with a higher ID value.

Example

```
UINT nTaskId;  
  
// Begin a file transfer in the background
```

```
nTaskId = HttpAsyncGetFile(hClient,
                           lpszLocalFile,
                           lpszRemoteFile,
                           HTTP_TRANSFER_DEFAULT,
                           0, NULL, 0);

if (nTaskId != 0)
{
    DWORD dwError = NO_ERROR;
    DWORD dwElapsed = 0;

    // Wait for the transfer to complete
    HttpTaskWait(nTaskId, INFINITE, &dwElapsed, &dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAsyncGetData](#), [HttpAsyncPutData](#), [HttpAsyncPutFile](#), [HttpEventProc](#), [HttpGetFile](#),
[HttpTaskDone](#), [HttpTaskWait](#)

HttpAsyncGetFileEx Function

```
UINT WINAPI HttpAsyncGetFileEx(  
    HCLIENT hClient,  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszRemoteFile,  
    DWORD dwOptions,  
    ULARGE_INTEGER uiOffset,  
    HTTPEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

Downloads the specified file from the server to the local system. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A pointer to a string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A pointer to a string that specifies the file on the server that will be transferred to the local system. The file naming conventions must be that of the host operating system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Constant	Description
HTTP_TRANSFER_DEFAULT	The default transfer mode. The resource data is copied to the local system exactly as it is stored on the server.
HTTP_TRANSFER_CONVERT	If the resource being downloaded from the server is textual, the data is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences.
HTTP_TRANSFER_COMPRESS	This option informs the server that the client is willing to accept compressed data. If the server supports compression for the specified resource, then the data will be automatically expanded before being returned to the caller. This option is selected by default if compression has been enabled using the HttpEnableCompression function. This option is ignored if the <i>dwOffset</i> parameter is non-zero.

uiOffset

A byte offset which specifies where the file transfer should begin. The default value of zero

specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **HttpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **HttpAsyncGetFileEx** function is used to download the contents of a remote file to a file on the local system. This function is similar to the **HttpGetFileEx** function, however it retrieves the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being transferred to the local system.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the HTTP_EVENT_CONNECT event after the connection has been established, and the HTTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated. During the file transfer, the callback function will be invoked periodically with the HTTP_EVENT_PROGRESS event. The client session handle is passed to the event handler, allowing you to call functions such as **HttpGetTransferStatusEx** to determine the amount of data that has been copied.

To determine when the transfer has completed without implementing an event handler, periodically call the **HttpTaskDone** function. If you wish to block the current thread and wait for the transfer to complete, call the **HttpTaskWait** function. To stop a background file transfer that is in progress, call the **HttpTaskAbort** function. This will signal the background worker thread to cancel the transfer and terminate the session.

This function can be called multiple times to download multiple files in the background; however, most servers limit the number of simultaneous connections that can originate from a single IP address. It is recommended that you only perform two simultaneous background transfers from the same server at any one time. The application should not make any assumptions about the order in which multiple background transfers may complete or how they are sequenced. For example, it should never be assumed that a background task with a lower task ID will complete before a task with a higher ID value.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAsyncPutFileEx](#), [HttpEventProc](#), [HttpGetFileEx](#), [HttpPutFileEx](#), [HttpTaskDone](#), [HttpTaskWait](#)

HttpAsyncPostData Function

```
UINT WINAPI HttpAsyncPostData(  
    HCLIENT hClient,  
    LPCTSTR lpszResource,  
    LPVOID lpvBuffer,  
    DWORD cbBuffer,  
    LPVOID lpvResult,  
    LPDWORD lpcbResult,  
    DWORD dwOptions,  
    HTTPEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

Post data from a local buffer to a script executed on the server.

Parameters

hClient

Handle to the client session.

lpszResource

A pointer to a string that specifies the resource that the data will be posted to on the server. Typically this is the name of a script that will be executed. This string may specify a valid URL for the current server that the client is connected to.

lpvBuffer

A pointer to the data that will be provided to the script. This parameter may be NULL if the script does not require any additional data from the client.

cbBuffer

The number of bytes to copy from the buffer. If this *lpvBuffer* parameter is NULL, this value should be zero.

lpvResult

A pointer to a byte buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvBuffer* parameter. If the *lpvResult* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of data that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_POST_DEFAULT	The default post mode. The contents of the buffer are encoded and sent as standard form data. The data returned by the server is copied to the result buffer exactly as it is returned from the server.

HTTP_POST_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.
HTTP_POST_MULTIPART	The contents of the buffer being sent to the server consists of multipart form data and will be sent as-is without any encoding.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **HttpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **HttpAsyncPostData** function is used to submit data to a script that executes on the server and then copy the output from that script into a local buffer. This function is similar to the **HttpPostData** function, however it submits the data using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the data is being sent and the response from the server is being returned to the caller.

Because this function works asynchronously, it is important that the memory allocated for the *lpvBuffer* and *lpvResult* parameters will not be released before the asynchronous task completes. If you provide pointers to memory that is allocated on the stack, ensure that your code does not return from the function until the background task completes. This can be achieved by calling the **HttpTaskWait** function or periodically calling the **HttpTaskDone** function to determine if the operation has completed. If you wish to return from the calling function immediately, then you must dynamically allocate memory on the heap and free that memory after the task has completed and the data is no longer needed.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the HTTP_EVENT_CONNECT event after the connection has been established, and the HTTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated.

The *lpvResult* parameter may be specified in one of two ways, depending on the needs of the

application. It can either be a pre-allocated buffer large enough to store the contents of the server response or it can specify the address of a global memory handle that will contain the data. If it points to a pre-allocated buffer, the *lpcbResult* parameter must be initialized to the maximum number of bytes that can be copied into the buffer. If specifies the address of a global memory handle, then *lpcbResult* must be initialized to a value of zero. See the example code below.

If you wish to submit XML formatted data to the server, it is recommended that you use the **HttpAsyncPostXml** function instead.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;
UINT nTaskId;

// Store the script output into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter. Since the output from the
// script is textual, the HTTP_POST_CONVERT option is used

nTaskId = HttpAsyncPostData(hClient,
                           lpzResource,
                           lpParameters,
                           cbParameters,
                           &hgblBuffer,
                           &cbBuffer,
                           HTTP_POST_CONVERT,
                           NULL, 0);

if (nTaskId != 0)
{
    DWORD dwError = NO_ERROR;
    DWORD dwElapsed = 0;

    // Wait for the task to complete
    HttpTaskWait(nTaskId, INFINITE, &dwElapsed, &dwError);

    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAsyncGetData](#), [HttpAsyncPostJson](#), [HttpAsyncPostXml](#), [HttpEventProc](#), [HttpPostData](#),

HttpAsyncPostJson Function

```
UINT WINAPI HttpAsyncPostJson(  
    HCLIENT hClient,  
    LPCTSTR lpszResource,  
    LPCTSTR lpszJsonData,  
    LPVOID lpvResult,  
    LPDWORD lpcbResult,  
    DWORD dwOptions,  
    HTTPEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

Submit JSON formatted data to the server and return the response to the caller.

Parameters

hClient

Handle to the client session.

lpszResource

A pointer to a string that specifies the resource name that the JSON data will be submitted to. Typically this is the name of a script on the server.

lpszJsonData

A pointer to a string that specifies the JSON data that will be submitted to the server.

lpvResult

A pointer to a byte buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvBuffer* parameter. If the *lpvResult* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of data that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_POST_DEFAULT	The default post mode. The contents of the buffer are encoded and sent as standard form data. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
HTTP_POST_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the

server.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **HttpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **HttpAsyncPostJson** function is used to submit JSON formatted data to a script that executes on the server and then copy the output from that script into a local buffer. It automatically sets the correct content type and encoding required for submitting JSON data to a server, however it does not parse the JSON data itself to ensure that it is well-formed. Your application is responsible for ensuring that the JSON data that is being submitted to the server is formatted correctly. This function is similar to the **HttpPostJson** function, however it submits the data using a background worker thread and does not block the current working thread.

Because this function works asynchronously, it is important that the memory allocated for the *lpzJsonData* and *lpvResult* parameters will not be released before the asynchronous task completes. If you provide pointers to memory that is allocated on the stack, ensure that your code does not return from the function until the background task completes. This can be achieved by calling the **HttpTaskWait** function or periodically calling the **HttpTaskDone** function to determine if the operation has completed. If you wish to return from the calling function immediately, then you must dynamically allocate memory on the heap and free that memory after the task has completed and the data is no longer needed.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the HTTP_EVENT_CONNECT event after the connection has been established, and the HTTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated.

The *lpvResult* parameter may be specified in one of two ways, depending on the needs of the application. It can either be a pre-allocated buffer large enough to store the contents of the server response or it can specify the address of a global memory handle that will contain the data. If it points to a pre-allocated buffer, the *lpcbResult* parameter must be initialized to the maximum number of bytes that can be copied into the buffer. If specifies the address of a global memory handle, then *lpcbResult* must be initialized to a value of zero. See the example code below.

Example

```
HGLOBAL hgb1Buffer = (HGLOBAL)NULL;  
LPBYTE lpBuffer = (LPBYTE)NULL;  
DWORD cbBuffer = 0;
```

```

UINT nTaskId;

// Store the script output into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter. Since the output from the
// script is textual, the HTTP_POST_CONVERT option is used

nTaskId = HttpAsyncPostJson(hClient,
                           lpszResource,
                           lpszJsonData,
                           &hgblBuffer,
                           &cbBuffer,
                           HTTP_POST_CONVERT,
                           NULL, 0);

if (nTaskId != 0)
{
    DWORD dwError = NO_ERROR;
    DWORD dwElapsed = 0;

    // Wait for the task to complete
    HttpTaskWait(nTaskId, INFINITE, &dwElapsed, &dwError);

    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAsyncGetData](#), [HttpAsyncPostData](#), [HttpEventProc](#), [HttpPostJson](#), [HttpTaskWait](#)

HttpAsyncPostXml Function

```
UINT WINAPI HttpAsyncPostXml(  
    HCLIENT hClient,  
    LPCTSTR lpszResource,  
    LPCTSTR lpszXmlData,  
    LPVOID lpvResult,  
    LPDWORD lpcbResult,  
    DWORD dwOptions,  
    HTTPEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

Submit XML formatted data to the server and return the response to the caller.

Parameters

hClient

Handle to the client session.

lpszResource

A pointer to a string that specifies the resource name that the XML data will be submitted to. Typically this is the name of a script on the server.

lpszXmlData

A pointer to a string that specifies the XML data that will be submitted to the server.

lpvResult

A pointer to a byte buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvBuffer* parameter. If the *lpvResult* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of data that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_POST_DEFAULT	The default post mode. The contents of the buffer are encoded and sent as standard form data. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
HTTP_POST_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the

server.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **HttpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **HttpAsyncPostXml** function is used to submit XML formatted data to a script that executes on the server and then copy the output from that script into a local buffer. It automatically sets the correct content type and encoding required for submitting XML data to a server, however it does not parse the XML data itself to ensure that it is well-formed. Your application is responsible for ensuring that the XML data that is being submitted to the server is formatted correctly. This function is similar to the **HttpPostXml** function, however it submits the data using a background worker thread and does not block the current working thread.

Because this function works asynchronously, it is important that the memory allocated for the *lpSzXmlData* and *lpvResult* parameters will not be released before the asynchronous task completes. If you provide pointers to memory that is allocated on the stack, ensure that your code does not return from the function until the background task completes. This can be achieved by calling the **HttpTaskWait** function or periodically calling the **HttpTaskDone** function to determine if the operation has completed. If you wish to return from the calling function immediately, then you must dynamically allocate memory on the heap and free that memory after the task has completed and the data is no longer needed.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the HTTP_EVENT_CONNECT event after the connection has been established, and the HTTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated.

The *lpvResult* parameter may be specified in one of two ways, depending on the needs of the application. It can either be a pre-allocated buffer large enough to store the contents of the server response or it can specify the address of a global memory handle that will contain the data. If it points to a pre-allocated buffer, the *lpcbResult* parameter must be initialized to the maximum number of bytes that can be copied into the buffer. If specifies the address of a global memory handle, then *lpcbResult* must be initialized to a value of zero. See the example code below.

Example

```
HGLOBAL hgb1Buffer = (HGLOBAL)NULL;  
LPBYTE lpBuffer = (LPBYTE)NULL;  
DWORD cbBuffer = 0;
```

```

UINT nTaskId;

// Store the script output into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter. Since the output from the
// script is textual, the HTTP_POST_CONVERT option is used

nTaskId = HttpAsyncPostXml(hClient,
                          lpszResource,
                          lpszXmlData,
                          &hgblBuffer,
                          &cbBuffer,
                          HTTP_POST_CONVERT,
                          NULL, 0);

if (nTaskId != 0)
{
    DWORD dwError = NO_ERROR;
    DWORD dwElapsed = 0;

    // Wait for the task to complete
    HttpTaskWait(nTaskId, INFINITE, &dwElapsed, &dwError);

    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAsyncGetData](#), [HttpAsyncPostData](#), [HttpEventProc](#), [HttpPostXml](#), [HttpTaskWait](#)

HttpAsyncProxyConnect Function

```
HCLIENT WINAPI HttpAsyncProxyConnect(  
    UINT nProxyType,  
    LPCTSTR LpszProxyHost,  
    UINT nProxyPort,  
    LPCTSTR LpszProxyUser,  
    LPCTSTR LpszProxyPassword,  
    LPCTSTR LpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    DWORD dwVersion,  
    LPSECURITYCREDENTIALS LpCredentials,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **HttpAsyncProxyConnect** function is used to establish a connection with a proxy server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **HttpProxyConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

nProxyType

An unsigned integer which specifies the type of proxy that the client is connecting to. The supported proxy server types are as follows:

Constant	Description
HTTP_PROXY_NONE	A direct connection will be established with the server. When this value is specified the proxy parameters are ignored.
HTTP_PROXY_STANDARD	A standard connection is established through the specified proxy server, and all resource requests will be specified using a complete URL. This proxy type should be used with standard connections.
HTTP_PROXY_SECURE	A secure connection is established through the specified proxy server. This proxy type should be used with secure connections and the HTTP_OPTION_SECURE option should also be set via the <i>dwOptions</i> parameter.
HTTP_PROXY_WINDOWS	The configuration options for the current system should be used. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. These settings are the same proxy server

lpszProxyHost

A pointer to a string which specifies the proxy server host name or IP address. This argument is ignored if the proxy type is set to HTTP_PROXY_NONE or HTTP_PROXY_WINDOWS and no proxy configuration has been specified for the local system.

nProxyPort

The port number that the proxy server is listening for connections on. A value of zero specifies that the default port number 80 should be used. Note that in most cases, a proxy server is not configured to use the default port. This argument is ignored if the proxy type is set to HTTP_PROXY_NONE or HTTP_PROXY_WINDOWS and no proxy configuration has been specified for the local system.

lpszProxyUser

A pointer to a string which specifies the user name that will be used to authenticate the client session to the proxy server. If the server does not require user authentication, then a NULL pointer may be passed in this argument.

lpszProxyPassword

A pointer to a string which specifies the password that will be used to authenticate the client session to the proxy server. If the server does not require user authentication, then a NULL pointer may be passed in this argument.

lpszRemoteHost

A pointer to a string which specifies the name of the server to connect to through the proxy server. This may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 80. For secure connections, the default port number is 443.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested. If the server does not support the keep-alive option, the client will automatically reconnect when each resource is requested.

	Although it will not provide any performance benefits, this allows the option to be used with all servers.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_ERRORDATA	This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the proxy server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a

	connection using IPv6 regardless if this option has been specified.
HTTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

dwVersion

The requested protocol version used when sending requests to the server. The high word should specify the major version, and the low word should specify the minor version number. The HTTPVERSION macro can be used to create version value.

lpCredentials

Pointer to credentials structure [SECURITYCREDENTIALS](#). This parameter is only used if the HTTP_OPTION_SECURE option is specified for the connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields *dwSize*, *lpzCertStore*, and *lpzCertName* must be defined, while other fields may be left undefined. Set *dwSize* to the size of the **SECURITYCREDENTIALS** structure.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **HttpGetLastError**.

Remarks

If the HTTP_PROXY_WINDOWS proxy type is specified, then the proxy configuration for the local system is used. If no proxy server has been defined, then the proxy-related parameters will be ignored and the function will establish a connection directly to the server.

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
HTTP_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.

HTTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
HTTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
HTTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
HTTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
HTTP_EVENT_PROGRESS	The client is in the process of sending or receiving data from the server. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.
HTTP_EVENT_REDIRECT	This event is generated when a the server indicates that the requested resource has been moved to a new location. The new resource location may be on the same server, or it may be located on another server. Check the value of the Location header field to determine where the resource has been moved to.

To cancel asynchronous notification and return the client to a blocking mode, use the **HttpDisableEvents** function.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **HttpAttachThread** function.

Specifying the HTTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same

handle.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAsyncConnect](#), [HttpConnect](#), [HttpDisconnect](#), [HttpInitialize](#), [HttpProxyConnect](#)

HttpAsyncPutData Function

```
UINT WINAPI HttpAsyncPutData(  
    HCLIENT hClient,  
    LPCTSTR lpszFileName,  
    LPVOID lpvBuffer,  
    DWORD dwLength,  
    DWORD dwReserved,  
    HTTPEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

Copies the contents of the specified buffer to a file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

A pointer to a string that specifies the file on the server that will be created, overwritten or appended to. The file naming conventions must be that of the host operating system.

lpvBuffer

A pointer to the data that will be copied to the server and stored in the specified file.

dwLength

The number of bytes to copy from the buffer.

dwReserved

A reserved parameter. This value should always be zero.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **HttpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **HttpAsyncPutData** function is used to upload the contents of a local buffer to the server. This function is similar to the **HttpPutData** function, however it uses a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the data is being sent to the server.

Because this function works asynchronously, it is important that the memory allocated for the buffer is not released before the asynchronous task completes. If you provide a buffer that is

allocated on the stack, ensure that your code does not return from the function while the data is being uploaded. This can be achieved by calling the **HttpTaskWait** function or periodically calling the **HttpTaskDone** function to determine if the transfer has completed. If you wish to return from the calling function immediately, then you must dynamically allocate memory for the *lpvBuffer* parameter on the heap and free that memory after the task has completed and the data is no longer needed.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the HTTP_EVENT_CONNECT event after the connection has been established, and the HTTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the data transfer is about to begin, and immediately before the worker thread is terminated. The worker thread creates a secondary connection to the server with its own session handle. This ensures that the asynchronous operation will not interfere with the current client session. Your application can interact with this background worker thread using the client handle that is passed to the event handler.

If the *lpvBuffer* parameter is pointing to a Unicode string, it is important to note that the value of the *dwLength* parameter should specify the number of bytes, not the number of characters. When using UTF-16, each character is two bytes long and therefore the length of the buffer is effectively double the length of the string. Because Unicode strings can contain null characters, you must also set the current file type to FILE_TYPE_IMAGE prior to calling this function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAsyncGetData](#), [HttpAsyncGetFile](#), [HttpAsyncPutFile](#), [HttpEventProc](#), [HttpPutData](#), [HttpTaskWait](#)

HttpAsyncPutFile Function

```
UINT WINAPI HttpAsyncPutFile(  
    HCLIENT hClient,  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszRemoteFile,  
    DWORD dwOptions,  
    DWORD dwOffset,  
    HTTPEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

Uploads the specified file from the local system to the server.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A pointer to a string that specifies the file that will be transferred from the local system. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A pointer to a string that specifies the file on the server that will be created, overwritten or appended to. The file naming conventions must be that of the host operating system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Constant	Description
HTTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the remote file exists, it will be overwritten with the contents of the uploaded file.

dwOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error

information, call the **HttpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **HttpAsyncPutFile** function is used to upload the contents of a local file to the server. This function is similar to the **HttpPutFile** function, however it uploads the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being transferred to the server.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the HTTP_EVENT_CONNECT event after the connection has been established, and the HTTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated. During the file transfer, the callback function will be invoked periodically with the HTTP_EVENT_PROGRESS event. The client session handle is passed to the event handler, allowing you to call functions such as **HttpGetTransferStatus** to determine the amount of data that has been copied.

To determine when the transfer has completed without implementing an event handler, periodically call the **HttpTaskDone** function. If you wish to block the current thread and wait for the transfer to complete, call the **HttpTaskWait** function. To stop a background file transfer that is in progress, call the **HttpTaskAbort** function. This will signal the background worker thread to cancel the transfer and terminate the session.

This function can be called multiple times to upload multiple files in the background; however, most servers limit the number of simultaneous connections that can originate from a single IP address. It is recommended that you only perform two simultaneous background transfers from the same server at any one time. The application should not make any assumptions about the order in which multiple background transfers may complete or how they are sequenced. For example, it should never be assumed that a background task with a lower task ID will complete before a task with a higher ID value.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAsyncGetData](#), [HttpAsyncGetFile](#), [HttpAsyncPutData](#), [HttpEventProc](#), [HttpPutFile](#), [HttpTaskDone](#), [HttpTaskWait](#)

HttpAsyncPutFileEx Function

```
UINT WINAPI HttpAsyncPutFileEx(  
    HCLIENT hClient,  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszRemoteFile,  
    DWORD dwOptions,  
    ULARGE_INTEGER uiOffset,  
    HTTPEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

Uploads the specified file from the local system to the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A pointer to a string that specifies the file that will be transferred from the local system. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A pointer to a string that specifies the file on the server that will be created, overwritten or appended to. The file naming conventions must be that of the host operating system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Constant	Description
HTTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the remote file exists, it will be overwritten with the contents of the uploaded file.

uiOffset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique

asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **HttpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **HttpAsyncPutFileEx** function is used to upload the contents of a local file to the server. This function is similar to the **HttpPutFileEx** function, however it uploads the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being transferred to the server.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the HTTP_EVENT_CONNECT event after the connection has been established, and the HTTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated. During the file transfer, the callback function will be invoked periodically with the HTTP_EVENT_PROGRESS event. The client session handle is passed to the event handler, allowing you to call functions such as **HttpGetTransferStatusEx** to determine the amount of data that has been copied.

To determine when the transfer has completed without implementing an event handler, periodically call the **HttpTaskDone** function. If you wish to block the current thread and wait for the transfer to complete, call the **HttpTaskWait** function. To stop a background file transfer that is in progress, call the **HttpTaskAbort** function. This will signal the background worker thread to cancel the transfer and terminate the session.

This function can be called multiple times to upload multiple files in the background; however, most servers limit the number of simultaneous connections that can originate from a single IP address. It is recommended that you only perform two simultaneous background transfers from the same server at any one time. The application should not make any assumptions about the order in which multiple background transfers may complete or how they are sequenced. For example, it should never be assumed that a background task with a lower task ID will complete before a task with a higher ID value.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAsyncGetFileEx](#), [HttpEventProc](#), [HttpGetFileEx](#), [HttpPutFileEx](#), [HttpTaskDone](#), [HttpTaskWait](#)

HttpAsyncSubmitForm Function

```
UINT WINAPI HttpAsyncSubmitForm(  
    HCLIENT hClient,  
    HFORM hForm,  
    LPVOID lpvResult,  
    LPDWORD lpcbResult,  
    DWORD dwOptions,  
    HTTPEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

Submit the specified form to the server for processing and return the response to the caller.

Parameters

hClient

Handle to the client session.

hForm

Handle to the virtual form which contains the data to be submitted to the server.

lpvResult

A pointer to a byte buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvBuffer* parameter. If the *lpvResult* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of data that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_SUBMIT_DEFAULT	The default post mode. The contents of the buffer are encoded and sent as standard form data. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
HTTP_SUBMIT_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more

information about the callback function, see the description of the **HttpEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **HttpGetLastError** function. The application should treat the task ID as an opaque value and never make an assumption about the sequence in which IDs are assigned to a background task.

Remarks

The **HttpAsyncSubmitForm** function is used to submit form data to a script that executes on the server and then copy the output from that script into a local buffer. This function is similar to the **HttpSubmitForm** function, however it submits the data using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the data is being sent and the response from the server is being returned to the caller.

Because this function works asynchronously, it is important that the memory allocated for the *lpvResult* parameter will not be released before the asynchronous task completes. If you provide a buffer that is allocated on the stack, ensure that your code does not return from the function until the background task completes. This can be achieved by calling the **HttpTaskWait** function or periodically calling the **HttpTaskDone** function to determine if the operation has completed. If you wish to return from the calling function immediately, then you must dynamically allocate the buffer on the heap and free that memory after the task has completed and the data is no longer needed.

Do not call the **HttpDestroyForm** function to release the memory allocated for the virtual form until the background task completes. Doing so may result in unpredictable behavior and could cause your application to terminate abnormally.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the HTTP_EVENT_CONNECT event after the connection has been established, and the HTTP_EVENT_DISCONNECT event after the transfer has completed. This enables your application to know when the file transfer is about to begin, and immediately before the worker thread is terminated.

The *lpvResult* parameter may be specified in one of two ways, depending on the needs of the application. It can either be a pre-allocated buffer large enough to store the contents of the server response or it can specify the address of a global memory handle that will contain the data. If it points to a pre-allocated buffer, the *lpcbResult* parameter must be initialized to the maximum number of bytes that can be copied into the buffer. If specifies the address of a global memory handle, then *lpcbResult* must be initialized to a value of zero. See the example code below.

Example

```
HFORM hForm = INVALID_FORM;  
HGLOBAL hgb1Result = (HGLOBAL)NULL;  
DWORD cbResult = 0;  
INT nResult = 0;  
UINT nTaskId;
```

```

hForm = HttpCreateForm(_T("/login.php"), HTTP_METHOD_POST, HTTP_FORM_ENCODED);

if (hForm == INVALID_FORM)
    return;

HttpAddFormField(hForm, _T("UserName"), lpszUserName, (DWORD)-1L, 0);
HttpAddFormField(hForm, _T("Password"), lpszPassword, (DWORD)-1L, 0);

nTaskId = HttpAsyncSubmitForm(hClient, hForm, &hgblResult, &cbResult, 0);

if (nTaskId != 0)
{
    DWORD dwError = NO_ERROR;
    DWORD dwElapsed = 0;

    // Wait for the task to complete
    HttpTaskWait(nTaskId, INFINITE, &dwElapsed, &dwError);

    // Get a pointer to the data returned by the server
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblResult);

    // lpBuffer now points to data returned by the server after
    // the form data was submitted

    // Release the memory allocated for the buffer
    GlobalUnlock(hgblResult);
    GlobalFree(hgblResult);
}

// Release the memory allocated for the virtual form
HttpDestroyForm(hForm);

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAddFormField](#), [HttpAddFormFile](#), [HttpClearForm](#), [HttpCreateForm](#), [HttpDeleteFormField](#), [HttpRegisterEvent](#)

HttpAttachThread Function

```
DWORD WINAPI HttpAttachThread(  
    HCLIENT hClient  
    DWORD dwThreadId  
);
```

The **HttpAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **HttpAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **HttpAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **HttpCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **HttpUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttp10.lib

See Also

[HttpCancel](#), [HttpAsyncConnect](#), [HttpConnect](#), [HttpDisconnect](#), [HttpUninitialize](#)

HttpAuthenticate Function

```
INT WINAPI HttpAuthenticate(  
    HCLIENT hClient,  
    UINT nAuthType,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword  
);
```

Parameters

hClient

Handle to the client session.

nAuthType

An unsigned integer value which specifies the method to be used when authenticating the client. The following values are recognized.

Constant	Description
HTTP_AUTH_NONE	No client authentication should be performed. The <i>lpszUserName</i> and <i>lpszPassword</i> parameters are ignored and current authentication settings are cleared.
HTTP_AUTH_BASIC	The Basic authentication scheme should be used. This option is supported by all servers that support at least version 1.0 of the protocol. The user credentials are not encrypted and Basic authentication should not be used over standard (non-secure) connections. Most web services which use Basic authentication require the connection to be secure.
HTTP_AUTH_BEARER	The Bearer authentication scheme should be used. This authentication method does not require a user name and the <i>lpszPassword</i> parameter must specify the OAuth 2.0 bearer token issued by the service provider. If the access token has expired, the request will fail with an authorization error. This function will not automatically refresh an expired token.

lpszUserName

A null terminated string which specifies the user name used to authenticate the client session. This parameter may be NULL or an empty string if a user name is not required for the specified authentication type.

lpszPassword

A pointer to a string that specifies the password used to authenticate the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

This function will set the **Authorization** request header for the client session using the credentials provided by the caller. This function will always override any custom **Authorization** header value that may have been previously set using the **HttpSetRequestHeader** function.

If both the *lpszUserName* and *lpszPassword* parameters are NULL pointers or specify zero length strings, the current authentication type will always be set to HTTP_AUTH_NONE regardless of the value of the *nAuthType* parameter. This effectively clears the current user credentials for the client session.

If the web service requires OAuth 2.0 authentication, it is recommended you use the **HttpSetBearerToken** function to specify the access token.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAsyncConnect](#), [HttpConnect](#), [HttpSetBearerToken](#), [HttpSetRequestHeader](#)

HttpCancel Function

```
INT WINAPI HttpCancel(  
    HCLIENT hClient  
);
```

The **HttpCancel** function cancels any outstanding blocking operation in the client, causing the blocking function to fail. The application may then retry the operation or terminate the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

When the **HttpCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

See Also

[HttpIsBlocking](#)

HttpClearForm Function

```
INT WINAPI HttpClearForm(  
    HFORM hForm  
);
```

The **HttpClearForm** function clears the specified form, removing all fields.

Parameters

hForm

Handle to the virtual form.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

See Also

[HttpAddFormField](#), [HttpAddFormFile](#), [HttpCreateForm](#), [HttpDeleteFormField](#), [HttpDestroyForm](#), [HttpSubmitForm](#)

HttpCloseFile Function

```
INT WINAPI HttpCloseFile(  
    HCLIENT hClient  
);
```

The **HttpCloseFile** function flushes the internal client buffers and closes the previously opened file on the server.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

If the file is opened for writing, all buffered data is written to the server before the file is closed. This may cause the client to block until all of the data can be written. The client application should not perform any other action until the function returns.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

See Also

[HttpCommand](#), [HttpGetData](#), [HttpGetFile](#), [HttpOpenFile](#), [HttpPutData](#), [HttpPutFile](#)

HttpCommand Function

```
INT WINAPI HttpCommand(  
    HCLIENT hClient,  
    LPCTSTR lpszCommand,  
    LPCTSTR lpszResource,  
    LPBYTE lpParameter,  
    DWORD cbParameter,  
    DWORD dwReserved  
);
```

The **HttpCommand** function sends a command to the server and returns the result code back to the caller. This function is typically used for extended commands not directly supported by the API.

Parameters

hClient

Handle to the client session.

lpszCommand

A pointer to a string which specifies the command to be executed by the server. The following table lists the standard commands recognized by most HTTP servers. Other commands may also be used, such as those extensions used by WebDAV to edit and manage files on a server.

Command	Description
GET	Return the contents of the specified resource. This command is recognized by all servers.
HEAD	Return only header information for the specified resource. This command is recognized by servers that support at least version 1.0 of the protocol.
POST	Post data to the specified resource. This command is recognized by servers that support at least version 1.0 of the protocol.
PUT	Create or replace the specified resource on the server. This command is recognized by servers that support at least version 1.0 of the protocol. Not all servers support this command.
DELETE	Delete the specified resource from the server. This command is recognized by servers that support at least version 1.1 of the protocol. Not all servers support this command.

lpszResource

A pointer to a string which specifies the resource to be used with the command. This can be the name of a file, an executable script or any other valid resource name recognized by the server. Resource names must be absolute and include the complete path to the resource.

lpParameter

A pointer to a byte array which contains data that is to be passed to the command as one or more parameters. Typically this is used to pass additional information to a script that executes on the server. The data is encoded according to the encoding type specified for the client session. If the resource does not require any parameters, this value should be NULL.

cbParameter

Specifies the number of bytes stored in the parameter buffer. If the resource does not require

any parameters, this value should be zero.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is the result code returned by the server. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

Not all servers support all of the listed commands, and some commands may require specific changes to the server configuration. In particular, the PUT and DELETE commands typically require that configuration changes be made by the site administrator. All servers will support the use of the GET command, and all servers that support at least version 1.0 of the protocol will support the POST command.

If the *lpSzResource* parameter specifies a path that contains reserved or restricted characters, such as a space, it will automatically be URL encoded by the library.

It is permissible to include a query string in the resource name specified by the *lpSzResource* parameter. Query strings begin with a question mark, and then are followed by one or more name/value pairs separated by an equal sign. For example, the following resource includes a query string:

```
/cgi-bin/test.cgi?field1=value1&field2=value2
```

In this case, the query string is "?field1=value1&field2=value2". If the query string contains reserved or restricted characters, such as spaces, then it will be automatically URL encoded prior to being sent to the server. If additional resource data is specified in the *lpParameter* argument along with a query string in the resource name, the action taken by the library depends on the command being sent. If the command is a POST or PUT command, then query string is included with command request to the server and the parameter data is sent separately. For example, if the POST command was used, the script running on the server would see that both query data and form data has been provided to it. However, if any other command is specified, the parameter data is simply appended to the query string.

The *lpParameter* argument is used to pass additional information to the server when a resource is requested. This is most commonly used to provide information to scripts, similar to how arguments are used when executing a program from the command line. Unless the POST command is being executed, the data in the buffer will automatically be encoded using the current encoding mechanism specified for the client. By default, the data is URL encoded, which means that any spaces and non-printable characters are converted to printable characters before submitted to the server. The type of encoding that is performed can be set by calling the **HttpSetEncodingType** function. Although the default encoding is appropriate for most applications, those that submit XML formatted data may need to change the encoding type.

Only one request may be in progress at one time for each client session. Use the **HttpCloseFile** function to terminate the request after all of the data has been read from the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttp10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpCloseFile](#), [HttpCreateFile](#), [HttpGetData](#), [HttpGetFile](#), [HttpGetResultCode](#), [HttpGetResultString](#), [HttpOpenFile](#), [HttpPostData](#), [HttpPutData](#), [HttpPutFile](#)

HttpConnect Function

```
HCLIENT WINAPI HttpConnect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    DWORD dwVersion,  
    LPSECURITYCREDENTIALS lpCredentials  
);
```

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used. For standard connections, the default port number is 80. For secure connections, the default port number is 443.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested. If the server does not support the keep-alive option, the client will automatically reconnect when each resource is requested. Although it will not provide any performance benefits, this allows the option to be used with all servers.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the

	<p>redirected resource will be located on a different server.</p>
HTTP_OPTION_PROXY	<p>This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed using the system Control Panel.</p>
HTTP_OPTION_ERRORDATA	<p>This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.</p>
HTTP_OPTION_TUNNEL	<p>This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.</p>
HTTP_OPTION_TRUSTEDSITE	<p>This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.</p>
HTTP_OPTION_SECURE	<p>This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.</p>
HTTP_OPTION_SECURE_FALLBACK	<p>This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.</p>
HTTP_OPTION_PREFER_IPV6	<p>This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.</p>

HTTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

dwVersion

The requested protocol version used when sending requests to the server. The high word should specify the major version, and the low word should specify the minor version number. The HTTPVERSION macro can be used to create version value.

lpCredentials

Pointer to credentials structure [SECURITYCREDENTIALS](#). This parameter is only used if the HTTP_OPTION_SECURE option is specified for the connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields *dwSize*, *lpzCertStore*, and *lpzCertName* must be defined, while other fields may be left undefined. Set *dwSize* to the size of the **SECURITYCREDENTIALS** structure.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **HttpGetLastError**.

Remarks

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **HttpAttachThread** function.

Specifying the HTTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpDisconnect](#), [HttpInitialize](#), [HttpProxyConnect](#)

HttpConnectUrl Function

```
HCLIENT WINAPI HttpConnectUrl(  
    LPCTSTR lpszURL,  
    UINT nTimeout,  
    DWORD dwOptions  
);
```

The **HttpConnectUrl** function establishes a connection with the specified server using a URL.

Parameters

lpszURL

A pointer to a string which specifies the URL for the server. The URL must follow the conventions for the Hypertext Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested. If the server does not support the keep-alive option, the client will automatically reconnect when each resource is requested. Although it will not provide any performance benefits, this allows the option to be used with all servers.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server,

	then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed using the system Control Panel.
HTTP_OPTION_ERRORDATA	This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
HTTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple

	threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is `INVALID_CLIENT`. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpConnectUrl** function uses an HTTP URL to establish a connection with a server. The URL must be in the following format:

```
[http|https]://[username : password] @[remotehost] [:remoteport] / [path / ...] [filename]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "http" scheme is used and the `HTTP_OPTION_SECURE` option is specified, that option will be ignored. To establish a secure connection, the "https" scheme must be specified. The **HttpValidateUrl** function can be used to verify that a URL is valid prior to calling this function.

The **HttpConnectUrl** function is designed to provide a simpler, more convenient interface to establishing a connection with a server. However, complex connections such as those using a proxy server or a secure connection which uses a client certificate will require the program to use the lower-level connection functions. If you only need to upload or download a file using a URL, then refer to the **HttpUploadFile** and **HttpDownloadFile** functions.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **HttpAttachThread** function.

Specifying the `HTTP_OPTION_FREETHREAD` option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Example

```
HCLIENT hClient;
LPCTSTR lpszURL = _T("http://sockettools.com/");

// Connect to the site specified by the URL
hClient = HttpConnectUrl(lpszURL, HTTP_TIMEOUT, HTTP_OPTION_DEFAULT);

if (hClient == INVALID_CLIENT)
{
    TCHAR szError[128];
```

```
// Display a message box that describes the error
HttpGetErrorString(HttpGetLastError(), szError, 128);
MessageBox(NULL, szError, NULL, MB_ICONEXCLAMATION|MB_TASKMODAL);
return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpConnect](#), [HttpDisconnect](#), [HttpDownloadFile](#), [HttpUploadFile](#), [HttpValidateUrl](#)

HttpCreateFile Function

```
INT WINAPI HttpCreateFile(  
    HCLIENT hClient,  
    LPCTSTR lpszRemoteFile,  
    DWORD dwLength,  
    DWORD dwOffset  
);
```

The **HttpCreateFile** function creates the specified file on the server.

Parameters

hClient

Handle to the client session.

lpszRemoteFile

Points to a string that specifies the name of the file being created on the server. The client must have the appropriate access rights to create the file or an error will be returned.

dwLength

Specifies the length of the file that will be created on the server. This value must be greater than zero.

dwOffset

Specifies a byte offset in the file. If this value is greater than zero, the server must support the ability to specify a byte range with the request to create the file, otherwise this function will fail.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpCreateFile** function uses the PUT command to create the file. The server must support this command and the user must have the appropriate permission to create the specified file. If this function is successful, the client should then use the **HttpWrite** function to send the contents of the file to the server. Once all of the data has been written, the **HttpCloseFile** function should be called to close the file and complete the operation. Note that this function is typically only accepted by servers that support version 1.1 of the protocol or later.

When using **HttpWrite** to send the contents of the file to the server, it is recommended that the data be written in logical blocks that are no larger than 8,192 bytes in size. Attempting to write very large amounts of data in a single call can either cause the thread to block or, in the case of an asynchronous connection, return an error if the internal buffers cannot accommodate all of the data. To send the entire contents of a file in a single function call, use the **HttpPutData** function instead of calling **HttpCreateFile**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpCloseFile](#), [HttpOpenFile](#), [HttpPutData](#), [HttpPutFile](#), [HttpWrite](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

HttpCreateFileEx Function

```
INT WINAPI HttpCreateFileEx(  
    HCLIENT hClient,  
    LPCTSTR lpszRemoteFile,  
    ULARGE_INTEGER uiLength,  
    ULARGE_INTEGER uiOffset  
);
```

The **HttpCreateFileEx** function creates the specified file on the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszRemoteFile

Points to a string that specifies the name of the file being created on the server. The client must have the appropriate access rights to create the file or an error will be returned.

uiLength

Specifies the length of the file that will be created on the server. This value must be greater than zero.

uiOffset

Specifies a byte offset into the file. If this value is greater than zero, the server must support the ability to specify a byte range with the request to create the file, otherwise this function will fail.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpCreateFileEx** function uses the PUT command to create the file. The server must support this command and the user must have the appropriate permission to create the specified file. If this function is successful, the client should then use the **HttpWrite** function to send the contents of the file to the server. Once all of the data has been written, the **HttpCloseFile** function should be called to close the file and complete the operation. Note that this function is typically only accepted by servers that support version 1.1 of the protocol or later.

When using **HttpWrite** to send the contents of the file to the server, it is recommended that the data be written in logical blocks that are no larger than 8,192 bytes in size. Attempting to write very large amounts of data in a single call can either cause the thread to block or, in the case of an asynchronous connection, return an error if the internal buffers cannot accommodate all of the data. To send the entire contents of a file in a single function call, use the **HttpPutData** function instead of calling **HttpCreateFileEx**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpCloseFile](#), [HttpOpenFile](#), [HttpPutData](#), [HttpPutFile](#), [HttpWrite](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

HttpCreateForm Function

```
HFORM WINAPI HttpCreateForm(  
    LPCTSTR lpszAction,  
    UINT nFormMethod,  
    UINT nFormType,  
    DWORD dwReserved  
);
```

The **HttpCreateForm** function creates a new form and returns a handle for use with the other form-related functions.

Parameters

lpszAction

A pointer to a string which specifies the name of the resource that the form data will be submitted to. Typically this is the name of a script that is executed on the server.

nFormMethod

An unsigned integer value which specifies how the form data will be submitted to the server. This parameter may be one of the following values:

Constant	Description
HTTP_METHOD_DEFAULT	The form data should be submitted using the default method, using the GET command.
HTTP_METHOD_GET	The form data should be submitted using the GET command. This method should be used when the amount of form data is relatively small. If the total amount of form data exceeds 2048 bytes, it is recommended that the POST method be used instead.
HTTP_METHOD_POST	The form data should be submitted using the POST command. This is the preferred method of submitting larger amounts of form data. If the total amount of form data exceeds 2048 bytes, it is recommended that the POST method be used.

nFormType

An unsigned integer value which specifies the type of form and how the data will be encoded when it is submitted to the server. This parameter may be one of the following values:

Constant	Description
HTTP_FORM_DEFAULT	The form data should be submitted using the default encoding method.
HTTP_FORM_ENCODED	The form data should be submitted as URL encoded values. This is typically used when the GET method is used to submit the data to the server.
HTTP_FORM_MULTIPART	The form data should be submitted as multipart form data. This is typically used when the POST method is used to submit a file to the server. Note that the script must understand how to process multipart form data if this form

type is specified.

dwReserved

A reserved parameter. This value must be zero.

Return Value

If the function succeeds, the return value is a handle to the virtual form. If the function fails, the return value is `INVALID_FORM`. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpCreateForm** function is used to create a new form that will be populated with values and then submitted to the server for processing. When the form is no longer needed, it should be destroyed using the **HttpDestroyForm** function.

Example

```
HFORM hForm = INVALID_FORM;
HGLOBAL hgblResult = (HGLOBAL)NULL;
DWORD cbResult = 0;
INT nResult = 0;

hForm = HttpCreateForm(_T("/login.php"), HTTP_METHOD_POST, HTTP_FORM_ENCODED);

if (hForm == INVALID_FORM)
    return;

HttpAddFormField(hForm, _T("UserName"), lpszUserName, (DWORD)-1L, 0);
HttpAddFormField(hForm, _T("Password"), lpszPassword, (DWORD)-1L, 0);

nResult = HttpSubmitForm(hClient, hForm, &hgblResult, &cbResult, 0);
HttpDestroyForm(hForm);

if (hgblResult != NULL)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblResult);

    // lpBuffer points to data returned by the server after the form
    // data was submitted

    GlobalUnlock(hgblResult);
    GlobalFree(hgblResult);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshttpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAddFormField](#), [HttpAddFormFile](#), [HttpClearForm](#), [HttpDeleteFormField](#), [HttpDestroyForm](#), [HttpSubmitForm](#)

HttpCreateSecurityCredentials Function

```
BOOL WINAPI HttpCreateSecurityCredentials(  
    DWORD dwProtocol,  
    DWORD dwOptions,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName,  
    LPVOID lpvReserved,  
    LPSECURITYCREDENTIALS* lppCredentials  
);
```

The **HttpCreateSecurityCredentials** function creates a **SECURITYCREDENTIALS** structure.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is

	supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A pointer to a string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A pointer to a string which specifies the certificate owner's password. A value of NULL specifies

that no password is required. This parameter is only used if a PKCS12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A pointer to a string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

lpvReserved

Pointer reserved for future use. Set it to NULL when using this function.

lppCredentials

Pointer to an [LPSECURITYCREDENTIALS](#) pointer. The memory for the credentials structure will be allocated by this function and must be released by calling the **HttpDeleteSecurityCredentials** function when it is no longer needed. The pointer value must be set to NULL before the function is called. It is important to note that this is a pointer to a pointer variable, not a pointer to the SECURITYCREDENTIALS structure itself.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The structure that is created by this function may be used as client credentials when establishing a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can

be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **HttpAsyncConnect** or **HttpConnect** functions.

Example

```
LPSECURITYCREDENTIALS lpSecCred = NULL;
HttpCreateSecurityCredentials(SEcurity_PROTOCOL_DEFAULT,
                             0,
                             NULL,
                             NULL,
                             lpzCertStore,
                             lpzCertName,
                             NULL,
                             &lpSecCred);

hClient = HttpConnect(lpzHostName,
                     HTTP_PORT_SECURE,
                     HTTP_TIMEOUT,
                     HTTP_OPTION_SECURE | HTTP_OPTION_KEEPALIVE,
                     HTTP_VERSION_11,
                     lpSecCred);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAsyncConnect](#), [HttpAsyncProxyConnect](#), [HttpConnect](#), [HttpDeleteSecurityCredentials](#), [HttpGetSecurityInformation](#), [HttpProxyConnect](#), [SECURITYCREDENTIALS](#)

HttpDeleteFile Function

```
INT WINAPI HttpDeleteFile(  
    HCLIENT hClient,  
    LPCTSTR lpszFileName  
);
```

The **HttpDeleteFile** function deletes the specified file from the server.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file to delete. The file pathing and name conventions must be that of the server.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

This function uses the DELETE command to delete the specified file from the server. The server must be configured to support this command, and client must have the appropriate permission to delete the file, or an error will be returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetFile](#), [HttpPutFile](#)

HttpDeleteFormField Function

```
INT WINAPI HttpDeleteFormField(  
    HFORM hForm,  
    LPCTSTR lpszFieldName  
);
```

The **HttpDeleteFormField** function deletes the specified field from the form.

Parameters

hForm

Handle to the virtual form.

lpszFieldName

Points to a string that specifies the name of the field to delete.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAddFormField](#), [HttpAddFormFile](#), [HttpClearForm](#), [HttpCreateForm](#), [HttpSubmitForm](#)

HttpDeleteHeaders Function

```
BOOL WINAPI HttpDeleteHeaders(  
    HCLIENT hClient,  
    UINT nHeaderType  
);
```

Delete all of the response or request headers for the current session.

Parameters

hClient

Handle to the client session.

nHeaderType

Specifies the type of header to delete. It may be one of the following values:

Constant	Description
HTTP_HEADERS_REQUEST	Delete all of the request headers that have been set for the next request sent to the server. This will clear all of the header values that were set using the HttpSetRequestHeader function.
HTTP_HEADERS_RESPONSE	Delete all of the headers that were set in response to the previous request. A call to the HttpGetResponseHeader function to obtain a specific header value will fail after this function returns.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpDeleteHeaders** function will release the memory allocated for the request or response headers for the current session. This function is typically used to clear all of the current request headers so that the application may create a new set of headers when a persistent connection is being used. The memory allocated for the request and response headers is normally released when the session handle is closed by calling the **HttpDisconnect** function.

Whenever a request for a resource is sent to the server, the response headers from the previous request are automatically cleared. It is not necessary for an application to call the **HttpDeleteHeaders** function to delete the response headers prior to each request.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttp10.lib

See Also

[HttpGetFirstHeader](#), [HttpGetNextHeader](#), [HttpGetResponseHeader](#), [HttpSetRequestHeader](#)

HttpDeleteSecurityCredentials Function

```
VOID WINAPI HttpDeleteSecurityCredentials(  
    LPSECURITYCREDENTIALS *LppCredentials  
);
```

The **HttpDeleteSecurityCredentials** function deletes an existing **SECURITYCREDENTIALS** structure.

Parameters

lppCredentials

Pointer to an **LPSECURITYCREDENTIALS** pointer. On exit from the function, the pointer will be NULL.

Return Value

None.

Example

```
if (lpSecCred)  
    HttpDeleteSecurityCredentials(&lpSecCred);  
HttpUninitialize();
```

Remarks

This function can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpCreateSecurityCredentials](#), [HttpUninitialize](#)

HttpDestroyForm Function

```
INT WINAPI HttpDestroyForm(  
    HFORM hForm  
);
```

The **HttpDestroyForm** function destroys the specified form, releasing the memory allocated for it.

Parameters

hForm

Handle to the form.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

See Also

[HttpAddFormField](#), [HttpAddFormFile](#), [HttpClearForm](#), [HttpCreateForm](#), [HttpDeleteFormField](#), [HttpSubmitForm](#)

HttpDisableEvents Function

```
INT WINAPI HttpDisableEvents(  
    HCLIENT hClient  
);
```

The **HttpDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

This function affects both event notification and event callbacks. Any outstanding events in the message queue should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpEnableEvents](#), [HttpFreezeEvents](#), [HttpRegisterEvent](#)

HttpDisableTrace Function

```
BOOL WINAPI HttpDisableTrace();
```

The **HttpDisableTrace** function disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpEnableTrace](#)

HttpDisconnect Function

```
INT WINAPI HttpDisconnect(  
    HCLIENT hClient  
);
```

The **HttpDisconnect** function terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAsyncConnect](#), [HttpConnect](#), [HttpUninitialize](#)

HttpDownloadFile Function

```
BOOL WINAPI HttpDownloadFile(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszFileURL,  
    UINT nTimeout  
    DWORD dwOptions  
    LPHTTPTRANSFERSTATUS lpStatus  
    HTTPEVENTPROC lpEventProc  
    DWORD_PTR dwParam  
);
```

The **HttpDownloadFile** function downloads the specified file from the server to the local system.

Parameters

lpszLocalFile

A pointer to a string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszFileURL

A pointer to a string that specifies the complete URL of the file to be downloaded. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed using the system Control Panel.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server

	certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.

lpStatus

A pointer to an HTTPTRANSFERSTATUS structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpDownloadFile** function provides a convenient way for an application to download a file in a single function call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session and then download the file to the local system. The URL must be complete, and specify either a standard or secure HTTP scheme:

```
[http|https]://[username : password] @] remotehost [:remoteport] / [path / ...] [filename]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "http" scheme is used and the HTTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, the "https" scheme must be specified.

The *lpStatus* parameter can be used by the application to determine the final status of the transfer, including the total number of bytes copied, the amount of time elapsed and other information related to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The *lpEventProc* parameter specifies a pointer to a function which will be periodically called during the file transfer process. This can be used to check the status of the transfer by calling **HttpGetTransferStatus** and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The *dwParam* parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the *this* pointer as the value, and then cast it back to an object pointer inside the callback function. If no event handler is required, then a NULL pointer can be specified as the value for *lpEventProc* and the *dwParam* parameter will be ignored.

The **HttpDownloadFile** function is designed to provide a simpler interface for downloading a file. However, complex connections such as those using a specific proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **HttpConnect** or **HttpProxyConnect** and then use **HttpGetFile** to download the file.

Example

```
HTTPTRANSFERSTATUS httpStatus;
LPCTSTR lpszLocalFile = _T("c:\\temp\\database.mdb");
LPCTSTR lpszFileURL = _T("http://www.example.com/updates/database.mdb");
BOOL bResult;

// Download the file using the specified URL
bResult = HttpDownloadFile(lpszLocalFile,
                          lpszFileURL,
                          HTTP_OPTION_PASSIVE,
                          &httpStatus,
                          NULL, 0);

if (!bResult)
{
    TCHAR szError[128];

    // Display a message box that describes the error
    HttpGetErrorString(HttpGetLastError(), szError, 128);
    MessageBox(NULL, szError, NULL, MB_ICONEXCLAMATION|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpEventProc](#), [HttpGetFile](#), [HttpGetTransferStatus](#), [HttpUploadFile](#), [HTTPTRANSFERSTATUS](#)

HttpDownloadFileEx Function

```
BOOL WINAPI HttpDownloadFileEx(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszFileURL,  
    UINT nTimeout  
    DWORD dwOptions  
    LPHTTPTRANSFERSTATUSEX lpStatus  
    HTTPEVENTPROC lpEventProc  
    DWORD_PTR dwParam  
);
```

The **HttpDownloadFileEx** function downloads the specified file from the server to the local system. This version of the function is designed to support files that are larger than 4GB.

Parameters

lpszLocalFile

A pointer to a string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszFileURL

A pointer to a string that specifies the complete URL of the file to be downloaded. The URL must follow the conventions for the File Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed using the system Control Panel.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.

HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.

lpStatus

A pointer to an HTTPTRANSFERSTATUSEX structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpDownloadFileEx** function provides a convenient way for an application to download a file in a single function call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session and then download the file to the local system. The URL must be complete, and specify either a standard or secure HTTP scheme:

```
[http|https]://[username : password] @] remotehost [:remoteport] / [path / ...] [filename]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "http" scheme is used and the HTTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, the "https" scheme must be specified.

The *lpStatus* parameter can be used by the application to determine the final status of the transfer, including the total number of bytes copied, the amount of time elapsed and other information related to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The *lpEventProc* parameter specifies a pointer to a function which will be periodically called during the file transfer process. This can be used to check the status of the transfer by calling **HttpGetTransferStatus** and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The *dwParam* parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the *this* pointer as the value, and then cast it back to an object pointer inside the callback function. If no event handler is required, then a NULL pointer can be

specified as the value for *lpEventProc* and the *dwParam* parameter will be ignored.

The **HttpDownloadFileEx** function is designed to provide a simpler interface for downloading a file. However, complex connections such as those using a specific proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **HttpConnect** or **HttpProxyConnect** and then use **HttpGetFile** to download the file.

Example

```
HTTPTRANSFERSTATUSEX httpStatus;
LPCTSTR lpszLocalFile = _T("c:\\temp\\database.mdb");
LPCTSTR lpszFileURL = _T("http://www.example.com/updates/database.mdb");
BOOL bResult;

// Download the file using the specified URL
bResult = HttpDownloadFileEx(lpszLocalFile,
                             lpszFileURL,
                             HTTP_OPTION_PASSIVE,
                             &httpStatus,
                             NULL, 0);

if (!bResult)
{
    TCHAR szError[128];

    // Display a message box that describes the error
    HttpGetErrorString(HttpGetLastError(), szError, 128);
    MessageBox(NULL, szError, NULL, MB_ICONEXCLAMATION|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpEventProc](#), [HttpGetFileEx](#), [HttpGetTransferStatusEx](#), [HttpUploadFileEx](#),
[HTTPTRANSFERSTATUSEX](#)

HttpEnableCompression Function

```
INT WINAPI HttpEnableCompression(  
    HCLIENT hClient,  
    BOOL bEnable  
);
```

The **HttpEnableCompression** function enables or disables support for data compression.

Parameters

hClient

Handle to the client session.

bEnable

A boolean value which specifies if data compression should be enabled or disabled. A non-zero value enables compression, while a value of zero will disable compression.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpEnableCompression** function is used to indicate to the server whether or not it is acceptable to compress the data that is returned to the client. If compression is enabled, the client will advertise that it will accept compressed data by setting the **Accept-Encoding** request header. The server will decide whether a resource being requested can be compressed. If the data is compressed, the library will automatically expand the data before returning it to the caller.

Enabling compression does not guarantee that the data returned by the server will actually be compressed, it only informs the server that the client is willing to accept compressed data. Whether or not a particular resource is compressed depends on the server configuration, and the server may decide to only compress certain types of resources, such as text files. Disabling compression informs the server that the client is not willing to accept compressed data; this is the default.

If the **HttpSetRequestHeader** function is used to explicitly set the **Accept-Encoding** header to request compressed data and compression is not enabled, the library will not attempt to automatically expand the data returned by the server. In this case, the raw compressed data will be returned to the caller and the application is responsible for processing it. This behavior is by design to maintain backwards compatibility with previous versions of the library that did not have internal support for compression.

To determine if the server compressed the data returned to the client, use the **HttpGetResponseHeader** function to get the value of the **Content-Encoding** header. If the header is defined, the value specifies the compression method used, otherwise the data was not compressed.

Enabling compression is only meaningful when downloading files from a server that supports file compression. It has no effect on file uploads.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

See Also

[HttpGetData](#), [HttpGetFile](#), [HttpGetFileEx](#), [HttpGetResponseHeader](#), [HttpGetText](#),
[HttpSetRequestHeader](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

HttpEnableEvents Function

```
INT WINAPI HttpEnableEvents(  
    HCLIENT hClient,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **HttpEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **HttpRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the event notification window. This window receives a user-defined message which specifies the event that has occurred. If this value is NULL, event notification is disabled.

uEventMsg

An unsigned integer which specifies the user-defined message that is sent when an event occurs. This parameter's value must be greater than the value of WM_USER. If the *hEventWnd* parameter is NULL, this value must be specified as WM_NULL.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
HTTP_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
HTTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
HTTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.

HTTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
HTTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
HTTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
HTTP_EVENT_PROGRESS	The client is in the process of sending or receiving data from the server. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.
HTTP_EVENT_REDIRECT	This event is generated when a the server indicates that the requested resource has been moved to a new location. The new resource location may be on the same server, or it may be located on another server. Check the value of the Location header field to determine where the resource has been moved to.

As noted, some events are only generated when the client is asynchronous mode. These events depend on the Windows Sockets asynchronous notification mechanism.

If event notification is disabled by specifying a NULL window handle, there may still be outstanding events in the message queue that must be processed. Since event handling has been disabled, these events should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

See Also

[HttpDisableEvents](#), [HttpFreezeEvents](#), [HttpRegisterEvent](#)

HttpEnableTrace Function

```
BOOL WINAPI HttpEnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **HttpEnableTrace** function enables the logging of Windows Sockets function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace function logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpDisableTrace](#)

HttpEnumTasks Function

```
INT WINAPI HttpEnumTasks(  
    UINT * lpTasks,  
    INT nMaxTasks,  
    DWORD dwOptions  
);
```

Return a list of active, suspended or finished asynchronous tasks.

Parameters

lpTasks

A pointer to an array of unsigned integer values that will contain unique task identifiers when the function returns. If this parameter is NULL, the function will return the number of tasks.

nMaxTasks

An integer value that specifies the maximum number of task identifiers that may be copied into the *lpTasks* array. If the *lpTasks* parameter is NULL, this value must be zero.

dwOptions

An unsigned integer that specifies the type of asynchronous tasks that may be returned by this function. It may be a combination of the following values:

Constant	Description
HTTP_TASK_DEFAULT	The list of asynchronous task IDs should include both active and suspended tasks. This option is the same as specifying both the HTTP_TASK_ACTIVE and HTTP_TASK_SUSPENDED options.
HTTP_TASK_ACTIVE	The list of asynchronous task IDs should include those tasks which are currently active. An active task represents a background connection to a server that is in the process of performing the requested action, such as uploading or downloading a file.
HTTP_TASK_SUSPENDED	The list of asynchronous task IDs should include those tasks which have been suspended. A suspended task represents a background connection that has been established, but the worker thread is not scheduled for execution.
HTTP_TASK_FINISHED	The list of asynchronous task IDs should include those tasks which have completed recently.

Return Value

If the function is successful, the return value is the number of task identifiers copied into the provided array. If there are no tasks which match the requested criteria, the return value is zero. A return value of HTTP_ERROR indicates an error has occurred. To get extended error information, call the **HttpGetLastError** function.

Remarks

The **HttpEnumTasks** function can be used to obtain a list of numeric identifiers that represent the asynchronous tasks that have been started or those that have completed. These task IDs are used by other functions to reference the background worker thread that has been created and obtain

status information for the task. For example, the **HttpTaskDone** function can be used to determine if a particular task has completed, and the **HttpTaskWait** function can be used to wait for a task to complete and return an error status code if the background operation failed.

There is an internal limit of 128 asynchronous tasks per process that may be active at any one time. When a task completes, the status information about that task is maintained for period of time after the task has completed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

See Also

[HttpTaskDone](#), [HttpTaskResume](#), [HttpTaskSuspend](#), [HttpTaskWait](#)

HttpEventProc Function

```
VOID CALLBACK HttpEventProc(  
    HCLIENT hClient,  
    UINT nEvent,  
    DWORD dwError,  
    DWORD_PTR dwParam  
);
```

The **HttpEventProc** function is an application-defined callback function that processes events generated by the client.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **HttpRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **HttpRegisterEvent** function. This callback function is also used by asynchronous tasks to notify the application when the task has started and completed. The **HttpEventProc** function is a placeholder for the application-defined function name.

If the callback function is invoked by an asynchronous task, it will execute in the context of the worker thread that is managing the client session. You must ensure that any access to global or static variables are synchronized, otherwise the results may be unpredictable. It is recommended that you do not declare any static variables within the callback function itself.

If the application has a graphical user interface, you should never attempt to directly modify a UI control from within the callback function for an asynchronous task. Controls should only be modified by the same UI thread that created their window. One common approach to resolve this issue is to post a user-defined message to the main window to signal that the user interface needs to be updated. The message handler would then process the user-defined message and update the user interface as needed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttp10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpDisableEvents](#), [HttpEnableEvents](#), [HttpFreezeEvents](#), [HttpRegisterEvent](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

HttpFreezeEvents Function

```
INT WINAPI HttpFreezeEvents(  
    HCLIENT hClient,  
    BOOL bFreeze  
);
```

The **HttpFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpDisableEvents](#), [HttpEnableEvents](#), [HttpRegisterEvent](#)

HttpGetBearerToken Function

```
INT WINAPI HttpGetBearerToken(  
    HCLIENT hClient,  
    LPTSTR lpszBearerToken,  
    INT nMaxLength  
);
```

The **HttpGetBearerToken** function returns the OAuth 2.0 bearer token used to authenticate the client session with a web service.

Parameters

hClient

Handle to the client session.

lpszBearerToken

A pointer to a string buffer which will contain the bearer token when the function returns. The string will be null terminated and the buffer must be large enough to accommodate the entire bearer token or the function will fail. This parameter cannot be a NULL pointer.

nMaxLength

The maximum number of characters that may be copied into the string buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the length of the bearer token string. A return value of zero indicates that no bearer token has been specified. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

This function returns the bearer token which was previously set by a call to **HttpSetBearerToken**. Bearer tokens can be very long strings and do not contain any human readable information. For Google services, these tokens are usually about 180 characters in length. For Microsoft services, their bearer tokens are typically 1,200 characters in length. It is recommended that you provide a buffer size of at least 2,000 characters.

Your application should not store the bearer tokens provided by a web service. These tokens are short-lived and typically only valid for about an hour. If the token has expired, the authorization to access the resource will fail and it must be refreshed. The refresh tokens used to acquire a new bearer token should be stored and they are typically valid for a period of months.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAuthenticate](#), [HttpSetBearerToken](#)

HttpGetCookie Function

```
BOOL WINAPI HttpGetCookie(  
    HCLIENT hClient,  
    LPCTSTR lpszCookieName,  
    LPTSTR lpszCookieValue,  
    INT nCookieValue,  
    LPTSTR lpszCookiePath,  
    INT nCookiePath,  
    LPTSTR lpszCookieDomain,  
    INT nCookieDomain,  
    LPSYSTEMTIME lpCookieExpires,  
    LPDWORD lpdwCookieFlags  
);
```

The **HttpGetCookie** function returns information about a cookie set by the server.

Parameters

hClient

Handle to the client session.

lpszCookieName

A pointer to a string which specifies the name of the cookie to return information about.

lpszCookieValue

A pointer to a string buffer that will contain the value of the cookie. If this information is not required, a NULL pointer may be specified.

nCookieValue

The maximum number of characters that may be copied into the buffer specified by the *lpszCookieValue* parameter, including the terminating null character.

lpszCookiePath

A pointer to a string buffer that will contain the cookie path. If this information is not required, a NULL pointer may be specified.

nCookiePath

The maximum number of characters that may be copied into the buffer specified by the *lpszCookiePath* parameter, including the terminating null character.

lpszCookieDomain

A pointer to a string buffer that will contain the cookie domain. If this information is not required, a NULL pointer may be specified.

nCookieDomain

The maximum number of characters that may be copied into the buffer specified by the *lpszCookieDomain* parameter, including the terminating null character.

lpCookieExpires

A pointer to a [SYSTEMTIME](#) structure which specifies the date and time that the cookie expires. If this information is not required, a NULL pointer may be specified.

lpdwCookieFlags

One or more bit flags which specify status information about the cookie. A value of zero indicates that there are no special status flags for the cookie. This parameter may be NULL if the

information is not required. The following values are currently defined:

Constant	Description
HTTP_COOKIE_SECURE	This flag specifies that the cookie should only be provided to the server if the connection is secure.
HTTP_COOKIE_SESSION	This flag specifies that the cookie should only be used for the current application session and should not be stored permanently on the local system.

Return Value

If the function succeeds, the return value is non-zero. If the specified cookie does not exist or function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The Hypertext Transfer Protocol uses special tokens called "cookies" to maintain persistent state information between requests for a resource. These cookies are exchanged between the client and server by setting specific header fields. When a server wants the client to use a cookie, it will include a header field named Set-Cookie in the response header when the client requests a resource. The client can then take this cookie and store it, either temporarily in memory or permanently in a file on the local system. The next time that the client requests a resource from that server, it can send the cookie back to the server by setting the Cookie header field. The **HttpGetCookie** function searches for a cookie set by the server in the Set-Cookie header field. The **HttpSetCookie** function creates or modifies the Cookie header field for the next resource requested by the client.

There are two general types of cookies that are used by servers. Session cookies exist only for the duration of the client session; they are stored in memory and not saved in any kind of permanent storage. When the client application terminates, session cookies are deleted and no longer used. Persistent cookies are stored on the local system and are used by the client until their expiration time. It is the responsibility of the client application to store persistent cookies; applications may use a flat text file, a database or any other storage method available.

In addition to the cookie name and value, the server may return additional information about the cookie which the client should use to determine if it should send the cookie back to the server:

The *cookie path* specifies a path for the resources where the cookie should be used. For example, a path of "/" indicates that the cookie should be provided for all resources requested from the server. A path of "/data" would mean that the cookie should be included if the resource is found in the /data folder or a sub-folder, such as /data/projections.asp. However, the cookie would not be provided if the resource /info/status.asp was requested, since it is not in the /data path.

The *cookie domain* specifies the domain for which the cookie should be used. Matches are made by comparing the name of the server against the domain name specified in the cookie. If the domain is example.com, then any server in the example.com domain would match; for example, both shipping.example.com and orders.example.com would match the domain value. However, if the cookie domain was orders.example.com, then the cookie would only be sent if the resource was requested from orders.example.com, not if the resource was located on shipping.example.com or www.example.com.

The *cookie expiration* specifies the date and time that the cookie should be deleted and no longer sent when a resource is requested from the server. This is only valid for persistent cookies, since session cookies are automatically deleted when the client

application terminates. The time is always expressed as Coordinated Universal Time.

The *cookie flags* provide additional information about the cookie. In some cases, a cookie should only be submitted to the server if the resource is requested using a secure connection. In this case, the bit flag HTTP_COOKIE_SECURE will be set.

It is the responsibility of the client application to determine if a cookie meets the criteria required to be submitted to the server. If the application wishes to send the cookie, it can use the **HttpSetCookie** function and specify the cookie name and value.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

See Also

[HttpGetFirstCookie](#), [HttpGetNextCookie](#), [HttpGetResponseHeader](#), [HttpSetCookie](#), [HttpSetRequestHeader](#)

HttpGetData Function

```
INT WINAPI HttpGetData(  
    HCLIENT hClient,  
    LPCTSTR lpszResource,  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwOptions  
);
```

The **HttpGetData** function requests a resource from the server and copies the data to the specified buffer.

Parameters

hClient

Handle to the client session.

lpszResource

A pointer to a string that specifies the resource on the server. This may be the name of a file on the server, or it may specify the name of a script that will be executed and the output returned to the caller. This string may specify a valid URL for the current server that the client is connected to.

lpvBuffer

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvBuffer* parameter. If the *lpvBuffer* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the file that was downloaded.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_TRANSFER_DEFAULT	The default transfer mode. The resource data is downloaded to the local system exactly as it is stored on the server. If you are requesting a text-based resource, the data may use a different end-of-line character sequence. For example, the end-of-line character may be a single linefeed character instead of a carriage return and linefeed pair.
HTTP_TRANSFER_CONVERT	If the resource being downloaded from the server is textual, the data is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences.

HTTP_TRANSFER_COMPRESS	This option informs the server that the client is willing to accept compressed data. If the server supports compression for the specified resource, then the data will be automatically expanded before being returned to the caller. This option is selected by default if compression has been enabled using the HttpEnableCompression function.
HTTP_TRANSFER_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail. Instead, the response is returned to the client and the function will succeed. If this option is used, your application should call HttpGetResultCode to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpGetData** function is used to retrieve a resource from the server and copy it into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the *lpvBuffer* parameter will point to the buffer that was allocated and the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvBuffer* parameter point to a global memory handle which will contain the file data when the function returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

If compression has been enabled and the server returns compressed data, it will be automatically expanded before being returned to the caller. This will result in a difference between the value returned in the *lpdwLength* parameter, which contains the actual number of bytes copied into the buffer, and the values reported by **HttpGetTransferStatus**. For example, if the server returns 5,000 bytes of compressed data that expands into 15,000 bytes, this function will return 15,000 as the number of bytes copied into the buffer. However, the **HttpGetTransferStatus** function will return the content length as the original 5,000 bytes of compressed data. For this reason, you should always use the value returned in the *lpdwLength* parameter to determine the amount of data that has been copied into the buffer.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **HttpGetTransferStatus** function.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Return the file data into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter
nResult = HttpGetData(hClient,
                    lpszResource,
                    &hgblBuffer,
                    &cbBuffer,
                    HTTP_TRANSFER_DEFAULT);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // resource data
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpEnableCompression](#), [HttpEnableEvents](#), [HttpGetFile](#), [HttpGetTransferStatus](#), [HttpPostData](#), [HttpPutData](#), [HttpPutFile](#), [HttpRegisterEvent](#)

HttpGetEncodingType Function

```
INT WINAPI HttpGetEncodingType(  
    HCLIENT hClient  
);
```

The **HttpGetEncodingType** function determines which content-encoding option is enabled.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is one of the following values:

Value	Constant	Description
0	HTTP_ENCODING_NONE	No encoding will be applied to the content of a request, and no Content-Type header will be generated.
1	HTTP_ENCODING_URL	URL encoding will be applied to the content of a request, and a Content-Type header will be generated with the value "application/x-www-form-urlencoded"
2	HTTP_ENCODING_XML	URL encoding will be applied to the content of a request, EXCEPT that spaces will not be replaced by '+'. This encoding type is intended for use with XML parsers that do not recognize '+' as a space. A Content-Type header will be generated with the value "application/x-www-form-urlencoded".

If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpCommand](#), [HttpPostData](#), [HttpSetEncodingType](#)

HttpGetErrorString Function

```
INT WINAPI HttpGetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);
```

The **HttpGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **HttpGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshttpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetLastError](#), [HttpSetLastError](#)

HttpGetFile Function

```
INT WINAPI HttpGetFile(  
    HCLIENT hClient,  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszRemoteFile,  
    DWORD dwOptions,  
    DWORD dwOffset  
);
```

The **HttpGetFile** function transfers the specified file on the server to the local system.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A pointer to a string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local system.

lpszRemoteFile

A pointer to a string that specifies the resource on the server. This may be the name of a file on the server, or it may specify the name of a script that will be executed and the output returned to the caller. This string may specify a valid URL for the current server that the client is connected to.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_TRANSFER_DEFAULT	The default transfer mode. The resource data is downloaded to the local system exactly as it is stored on the server. If you are requesting a text-based resource, the data may use a different end-of-line character sequence. For example, the end-of-line character may be a single linefeed character instead of a carriage return and linefeed pair.
HTTP_TRANSFER_CONVERT	If the resource being downloaded from the server is textual, the data is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences.
HTTP_TRANSFER_COMPRESS	This option informs the server that the client is willing to accept compressed data. If the server supports compression for the specified resource, then the data will be automatically expanded before being returned to the caller. This option is selected by default if compression has been enabled using the HttpEnableCompression function. This option is

	ignored if the <i>dwOffset</i> parameter is non-zero.
HTTP_TRANSFER_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail. Instead, the response is returned to the client and the function will succeed. If this option is used, your application should call HttpGetResultCode to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

dwOffset

Specifies a byte offset into the file. If this value is greater than zero, the server must support the ability to specify a byte range with the request to download the file, otherwise this function will fail.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

Enabling compression does not guarantee that the data returned by the server will actually be compressed, it only informs the server that the client is willing to accept compressed data. Whether or not a particular resource is compressed depends on the server configuration, and the server may decide to only compress certain types of resources, such as text files. To determine if the server compressed the data returned to the client, use the **HttpGetResponseHeader** function to get the value of the **Content-Encoding** header after this function returns. If the header is defined, the value specifies the compression method used, otherwise the data was not compressed.

To download large files that are over 4GB, use the **HttpGetFileEx** function.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **HttpGetTransferStatus** function.

Example

```
LPCTSTR lpszLocalFile = _T("index.html");
LPCTSTR lpszRemoteFile = _T("http://sockettools.com/");

if (HttpGetFile(hClient, lpszLocalFile, lpszRemoteFile) == HTTP_ERROR)
{
    dwError = HttpGetLastError();
    _tprintf(stderr, "Unable to download %s, error 0x%lx\n", lpszRemoteFile,
dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpEnableEvents](#), [HttpGetData](#), [HttpGetFileEx](#), [HttpGetText](#), [HttpGetTransferStatus](#), [HttpPutFile](#), [HttpRegisterEvent](#)

HttpGetFileEx Function

```
INT WINAPI HttpGetFileEx(  
    HCLIENT hClient,  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszRemoteFile,  
    DWORD dwOptions,  
    ULARGE_INTEGER uiOffset  
);
```

The **HttpGetFileEx** function transfers the specified file on the server to the local system. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A pointer to a string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A pointer to a string that specifies the resource on the server. This may be the name of a file on the server, or it may specify the name of a script that will be executed and the output returned to the caller. This string may specify a valid URL for the current server that the client is connected to.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_TRANSFER_DEFAULT	The default transfer mode. The resource data is downloaded to the local system exactly as it is stored on the server. If you are requesting a text-based resource, the data may use a different end-of-line character sequence. For example, the end-of-line character may be a single linefeed character instead of a carriage return and linefeed pair.
HTTP_TRANSFER_CONVERT	If the resource being downloaded from the server is textual, the data is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences.
HTTP_TRANSFER_COMPRESS	This option informs the server that the client is willing to accept compressed data. If the server supports compression for the specified resource, then the data will be automatically expanded before being returned to the caller. This option is selected by default if compression has been enabled using the

	HttpEnableCompression function. This option is ignored if the <i>uiOffset</i> parameter is non-zero.
HTTP_TRANSFER_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail. Instead, the response is returned to the client and the function will succeed. If this option is used, your application should call HttpGetResultCode to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

uiOffset

Specifies a byte offset into the file. If this value is greater than zero, the server must support the ability to specify a byte range with the request to download the file, otherwise this function will fail.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

Enabling compression does not guarantee that the data returned by the server will actually be compressed, it only informs the server that the client is willing to accept compressed data. Whether or not a particular resource is compressed depends on the server configuration, and the server may decide to only compress certain types of resources, such as text files. To determine if the server compressed the data returned to the client, use the **HttpGetResponseHeader** function to get the value of the **Content-Encoding** header after this function returns. If the header is defined, the value specifies the compression method used, otherwise the data was not compressed.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **HttpGetTransferStatusEx** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttp10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpEnableEvents](#), [HttpGetData](#), [HttpGetText](#), [HttpGetTransferStatusEx](#), [HttpPutFileEx](#), [HttpRegisterEvent](#)

HttpGetFileSize Function

```
INT WINAPI HttpGetFileSize(  
    HCLIENT hClient,  
    LPCTSTR lpszFileName,  
    LPDWORD lpdwFileSize  
);
```

The **HttpGetFileSize** function returns the size of the specified file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file.

lpdwFileSize

Points to an unsigned integer that will contain the size of the specified file in bytes.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

This function uses the HEAD command to retrieve header information about the file without downloading the contents of the file itself. This requires that the server support at least version 1.0 of the protocol standard, or an error will be returned.

The server may not return a file size for some resources. This is typically the case with scripts that generate dynamic content because the server has no way of determining the size of the output generated by the script without actually executing it. The server may also not provide a file size for HTML documents which use server side includes (SSI) because that content is also dynamically created by the server. If the request to the server was successful and the file exists, but the server does not return a file size, the function will succeed but the file size returned to the caller will be zero.

When a request is made to the server for information about the file, the library will attempt to keep the connection alive, even if the HTTP_OPTION_KEEPALIVE option has not been specified for the session. This allows an application to request the file size and then download the file without having to write additional code to re-establish the connection. However, it is possible that the attempt to keep the connection open will fail. In that case, an error will be returned and the client handle will no longer be valid. If this happens, the *lpdwFileSize* parameter may still contain a valid value. If the library was able to determine the file size, but was not able to maintain the connection to the server, the returned file size will be greater than zero even if the function returns an error.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttp10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpCommand](#), [HttpGetFileTime](#), [HttpVerifyFile](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

HttpGetFileSizeEx Function

```
INT WINAPI HttpGetFileSizeEx(  
    HCLIENT hClient,  
    LPCTSTR lpszFileName,  
    LPULARGE_INTEGER lpuiFileSize  
);
```

The **HttpGetFileSizeEx** function returns the size of the specified file on the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file.

lpdwFileSize

Points to a ULARGE_INTEGER structure that will contain the size of the specified file in bytes.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

This function uses the HEAD command to retrieve header information about the file without downloading the contents of the file itself. This requires that the server support at least version 1.0 of the protocol standard, or an error will be returned.

The server may not return a file size for some resources. This is typically the case with scripts that generate dynamic content because the server has no way of determining the size of the output generated by the script without actually executing it. The server may also not provide a file size for HTML documents which use server side includes (SSI) because that content is also dynamically created by the server. If the request to the server was successful and the file exists, but the server does not return a file size, the function will succeed but the file size returned to the caller will be zero.

When a request is made to the server for information about the file, the library will attempt to keep the connection alive, even if the HTTP_OPTION_KEEPALIVE option has not been specified for the session. This allows an application to request the file size and then download the file without having to write additional code to re-establish the connection. However, it is possible that the attempt to keep the connection open will fail. In that case, an error will be returned and the client handle will no longer be valid. If this happens, the *lpdwFileSize* parameter may still contain a valid value. If the library was able to determine the file size, but was not able to maintain the connection to the server, the returned file size will be greater than zero even if the function returns an error.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttp10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpCommand](#), [HttpGetFileTime](#), [HttpVerifyFile](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

HttpGetFileTime Function

```
INT WINAPI HttpGetFileTime(  
    HCLIENT hClient,  
    LPCTSTR lpszFileName,  
    LPSYSTEMTIME lpFileTime,  
    BOOL bLocalize  
);
```

The **HttpGetFileTime** function returns the modification time for the specified file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file.

lpFileTime

Points to a [SYSTEMTIME](#) structure that will be set to the current modification time for the remote file.

bLocalize

A boolean flag which specifies if the file time is localized to the current timezone. If this value is non-zero, then the file time is adjusted to that the time is local to the current system. If this value is zero, the file time is returned in UTC time.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpGetFileTime** function can be used to determine the date and time that a file was last modified on the server. The time may either be localized to the current system, or it may be returned as UTC time.

This function uses the HEAD command to retrieve header information about the file without downloading the contents of the file itself. This requires that the server support at least version 1.0 of the protocol standard, or an error will be returned.

The server may not return a modification time for some resources. If the request to the server was successful and the file exists, but the server does not return a modification time, the function will succeed but all of the members of the SYSTEMTIME structure will be zero.

When a request is made to the server for information about the file, the library will attempt to keep the connection alive, even if the HTTP_OPTION_KEEPALIVE option has not been specified for the session. This allows an application to request the modification time and then download the file without having to write additional code to re-establish the connection. However, it is possible that the attempt to keep the connection open will fail. In that case, an error will be returned and the client handle will no longer be valid. If this happens, the SYSTEMTIME structure may still contain a valid value. If the library was able to determine the modification time, but was not able to maintain the connection to the server, the members of the SYSTEMTIME structure will specify a valid date and time.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpCommand](#), [HttpGetFileSize](#)

HttpGetFirstCookie

```
DWORD WINAPI HttpGetFirstCookie(  
    HCLIENT hClient,  
    LPTSTR lpszCookieName,  
    LPINT lpnNameLen,  
    LPTSTR lpszCookieValue,  
    LPINT lpnValueLen  
);
```

The **HttpGetFirstCookie** function returns the first cookie set by the server.

Parameters

hClient

Handle to the client session.

lpszCookieName

A pointer to a string buffer which will contain the name of the first cookie set by the server.

lpnNameLen

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the function returns, this value is updated to specify the actual length of the cookie name string.

lpszCookieValue

A pointer to a string buffer which will contain the name of the first cookie value set by the server. If the cookie value is not required, this parameter may be NULL.

lpnValueLen

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the function returns, this value is updated to specify the actual length of the cookie value string. If the *lpszCookieValue* parameter is NULL, this parameter should also be NULL.

Return Value

If the function succeeds, the return value is a 32-bit token which will be passed to the **HttpGetNextCookie** function to retrieve the next cookie. If there are no cookies or the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpGetFirstCookie** function is used to enumerate the cookies set by the server after a resource has been requested. To get information about a specific cookie, use the **HttpGetCookie** function.

Example

```
TCHAR szCookieName[MAXCOOKIENAME];  
TCHAR szCookieValue[MAXCOOKIEVALUE];  
INT nNameLen, nValueLen;  
DWORD dwCookie;  
  
// Initialize the nNameLen and nValueLen variables to  
// the maximum number of characters that can be copied  
// into the string buffers  
nNameLen = MAXCOOKIENAME;
```

```

nValueLen = MAXCOOKIEVALUE;

// Get the first cookie set by the server
dwCookie = HttpGetFirstCookie(hClient,
                             szCookieName,
                             &nNameLen,
                             szCookieValue,
                             &nValueLen);

while (dwCookie != 0)
{
    // The szCookieName and szCookieValue strings contain the
    // the name and value for a cookie set by the server

    // Re-initialize the nNameLen and nValueLen variables
    // to the maximum length of the strings
    nNameLen = MAXCOOKIENAME;
    nValueLen = MAXCOOKIEVALUE;

    // Get the next cookie set by the server
    dwCookie = HttpGetNextCookie(hClient,
                                 dwCookie,
                                 szCookieName,
                                 &nNameLen,
                                 szCookieValue,
                                 &nValueLen);
};

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttp10.lib

See Also

[HttpGetCookie](#), [HttpGetNextCookie](#), [HttpGetResponseHeader](#), [HttpSetCookie](#),
[HttpSetRequestHeader](#)

HttpGetFirstHeader Function

```
BOOL WINAPI HttpGetFirstHeader(  
    HCLIENT hClient,  
    UINT nHeaderType,  
    LPTSTR lpszHeader,  
    LPINT lpcchHeader  
    LPTSTR lpszValue,  
    LPINT lpcchValue  
);
```

The **HttpGetFirstHeader** function returns the name and value of the first request or response header.

Parameters

hClient

Handle to the client session.

nHeaderType

Specifies the type of header to return. It may be one of the following values:

Constant	Description
HTTP_HEADERS_REQUEST	Return the first header set by the client before a resource has been requested. Request headers typically specify information that the server needs to complete the request, such as the name of the host that has the resource, or what types of resources are acceptable to the client.
HTTP_HEADERS_RESPONSE	Return the first header set by the server after a request has been submitted to the server. The response headers typically contain information about the resource, such as its size, the content type and the date it was last modified.

lpszHeader

A pointer to a string buffer that will contain the name of the header field when the function returns.

lpcchHeader

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the function returns, this value is updated to specify the actual length of the header name string.

lpszValue

A pointer to a string buffer that will contain the name of the header value when the function returns.

lpcchValue

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the function returns, this value is updated to specify the actual length of the header value string.

Return Value

If the function succeeds, the return value is non-zero. If the header field does not exist or the client handle is invalid, the function returns a value of zero. To get extended error information, call **HttpGetLastError**.

Remarks

Use this function together with **HttpGetNextHeader** to enumerate all request or response headers.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetResponseHeader](#), [HttpSetRequestHeader](#), [HttpGetNextHeader](#)

HttpGetFormProperties Function

```
INT WINAPI HttpGetFormProperties(  
    HFORM hForm,  
    LPHTTPFORMPROPERTIES lpFormProp  
);
```

The **HttpGetFormProperties** function returns information about the specified form.

Parameters

hForm

Handle to the virtual form.

lpFormProp

Points to a HTTPFORMPROPERTIES structure which will contain information about the specified form.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpCreateForm](#), [HttpSetFormProperties](#), [HttpSubmitForm](#), [HTTPFORMPROPERTIES](#)

HttpGetLastError Function

```
DWORD WINAPI HttpGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **HttpSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **HttpGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **HttpSetLastError(0)** when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or HTTP_ERROR. Those functions which call **HttpSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

See Also

[HttpGetErrorString](#), [HttpSetLastError](#)

HttpGetFirstCookie

```
DWORD WINAPI HttpGetNextCookie(  
    HCLIENT hClient,  
    DWORD dwCookie,  
    LPTSTR lpszCookieName,  
    LPINT lpnNameLen,  
    LPTSTR lpszCookieValue,  
    LPINT lpnValueLen  
);
```

The **HttpGetNextCookie** function returns the next cookie set by the server.

Parameters

hClient

Handle to the client session.

dwCookie

An unsigned integer value which specifies a cookie token returned by a previous call to either **HttpGetFirstCookie** or **HttpGetNextCookie**.

lpszCookieName

A pointer to a string buffer which will contain the name of the first cookie set by the server.

lpnNameLen

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the function returns, this value is updated to specify the actual length of the cookie name string.

lpszCookieValue

A pointer to a string buffer which will contain the name of the first cookie value set by the server. If the cookie value is not required, this parameter may be NULL.

lpnValueLen

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the function returns, this value is updated to specify the actual length of the cookie value string. If the *lpszCookieValue* parameter is NULL, this parameter should also be NULL.

Return Value

If the function succeeds, the return value is a 32-bit token which will be passed to the **HttpGetNextCookie** function to retrieve the next cookie. If there are no more cookies or the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpGetNextCookie** function is used to enumerate the cookies set by the server after a resource has been requested. To get information about a specific cookie, use the **HttpGetCookie** function.

Example

```
TCHAR szCookieName[MAXCOOKIENAME];  
TCHAR szCookieValue[MAXCOOKIEVALUE];  
INT nNameLen, nValueLen;  
DWORD dwCookie;
```

```

// Initialize the nNameLen and nValueLen variables to
// the maximum number of characters that can be copied
// into the string buffers
nNameLen = MAXCOOKIENAME;
nValueLen = MAXCOOKIEVALUE;

// Get the first cookie set by the server
dwCookie = HttpGetFirstCookie(hClient,
                              szCookieName,
                              &nNameLen,
                              szCookieValue,
                              &nValueLen);

while (dwCookie != 0)
{
    // The szCookieName and szCookieValue strings contain the
    // the name and value for a cookie set by the server

    // Re-initialize the nNameLen and nValueLen variables
    // to the maximum length of the strings
    nNameLen = MAXCOOKIENAME;
    nValueLen = MAXCOOKIEVALUE;

    // Get the next cookie set by the server
    dwCookie = HttpGetNextCookie(hClient,
                                 dwCookie,
                                 szCookieName,
                                 &nNameLen,
                                 szCookieValue,
                                 &nValueLen);
};

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

See Also

[HttpGetCookie](#), [HttpGetFirstCookie](#), [HttpGetResponseHeader](#), [HttpSetCookie](#),
[HttpSetRequestHeader](#)

HttpGetNextHeader Function

```
BOOL WINAPI HttpGetNextHeader(  
    HCLIENT hClient,  
    UINT nHeaderType,  
    LPTSTR lpszHeader,  
    LPINT lpcchHeader,  
    LPTSTR lpszValue,  
    LPINT lpcchValue  
);
```

The **HttpGetNextHeader** function returns the name and value of the next request or response header.

Parameters

hClient

Handle to the client session.

nHeaderType

Specifies the type of header to return. It may be one of the following values:

Constant	Description
HTTP_HEADERS_REQUEST	Return the first header set by the client before a resource has been requested. Request headers typically specify information that the server needs to complete the request, such as the name of the host that has the resource, or what types of resources are acceptable to the client.
HTTP_HEADERS_RESPONSE	Return the first header set by the server after a request has been submitted to the server. The response headers typically contain information about the resource, such as its size, the content type and the date it was last modified.

lpszHeader

A pointer to a string buffer that will contain the name of the header field when the function returns.

lpcchHeader

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the function returns, this value is updated to specify the actual length of the header name string.

lpszValue

A pointer to a string buffer that will contain the name of the header value when the function returns.

lpcchValue

A pointer to an integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. When the function returns, this value is updated to specify the actual length of the header value string.

Return Value

If the function succeeds, the return value is non-zero. If the header field does not exist or the client handle is invalid, the function returns a value of zero. To get extended error information, call **HttpGetLastError**.

Remarks

Use this function iteratively after **HttpGetFirstHeader** to enumerate all request or response headers.

Unlike the **HttpGetResponseHeader** function, which returns a single header name and value, the **HttpGetFirstHeader** and **HttpGetNextHeader** functions will return multiple headers that have the same common name.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetResponseHeader](#), [HttpSetRequestHeader](#), [HttpGetFirstHeader](#)

HttpGetOption Function

```
INT WINAPI HttpGetOption(  
    HCLIENT hClient,  
    DWORD dwOption,  
    LPBOOL lpbEnabled  
);
```

The **HttpGetOption** function determines whether a specified HTTP option is enabled.

Parameters

hClient

Handle to the client session.

dwOption

An unsigned integer which specifies the option that is to be checked. It may be one of the following values:

Constant	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_ERRORDATA	This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.
HTTP_OPTION_NOUSERAGENT	This option specifies the client should not include a User-Agent header with any requests made during the session. The user agent is a string which is used to identify the client application to the server.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP

	connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol. The client will default to using TLS 1.2 or later for secure connections.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
HTTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

lpbEnabled

A pointer to an integer which will be set to a non-zero value when the function returns if the specified option has been enabled. If the option has not been enabled, a zero value will be

returned in the variable.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**. Note that the value returned in the *lpbEnabled* parameter is only valid if the function succeeds.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

See Also

[HttpAsyncConnect](#), [HttpSetOption](#)

HttpGetPriority Function

```
INT WINAPI HttpGetPriority(  
    HCLIENT hClient  
);
```

The **HttpGetPriority** function returns a value which specifies the priority of file transfers.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the current file transfer priority. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpGetPriority** function can be used to determine the current priority assigned to file transfers performed by the client. It may be one of the following values:

Constant	Description
HTTP_PRIORITY_NORMAL	The default priority which balances resource utilization and transfer speed. It is recommended that most applications use this priority.
HTTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the transfer. It is typically used with worker threads running in the background when the amount of time required perform the transfer is not critical.
HTTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the transfer and meters the bandwidth allocated for the transfer. This priority will increase the average amount of time required to complete a file transfer.
HTTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the transfer, allocating more memory for internal buffering. It can be used when it is important to transfer the file quickly, and there are no other threads currently performing file transfers at the time.
HTTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization while attempting to transfer the file as quickly as possible. If the file transfer is being performed in the main UI thread, this priority can cause the application to appear to become non-responsive. No events will be generated during the transfer.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpSetPriority](#)

HttpGetRequestHeader Function

```
BOOL WINAPI HttpGetRequestHeader(  
    HCLIENT hClient,  
    LPCTSTR lpszHeader,  
    LPTSTR lpszValue,  
    INT nMaxLength  
);
```

The **HttpGetRequestHeader** function returns the value of the specified request header field.

Parameters

hClient

Handle to the client session.

lpszHeader

Pointer to a string which specifies the header value to be returned.

lpszValue

Pointer to a buffer which will contain the null-terminated string value of the specified header field.

nMaxLength

Maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is non-zero. If the header field does not exist or the client handle is invalid, the function returns a value of zero. To get extended error information, call **HttpGetLastError**.

Remarks

When a resource is requested from the server, it consists of two parts. The first part is the command issued to the server, along with the name of the resource and any optional encoded parameters. The second part consists of one or more header fields which can be used to provide additional information to the server. For example, here is what the command and header may look like for a simple request:

```
GET /test HTTP/1.0  
Host: api.sockettools.com  
Accept: text/*
```

The first line consists of the command, the resource and the protocol version. The subsequent lines are the header, which is similar to the headers used in email messages. The Host field specifies the name of the server the resource is being requested from, and the Accept field specifies the type of resources that are acceptable to the client; in this case, any type of text file.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetResponseHeader](#), [HttpSetRequestHeader](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

HttpGetResponseHeader Function

```
BOOL WINAPI HttpGetResponseHeader(  
    HCLIENT hClient,  
    LPCTSTR lpszHeader,  
    LPTSTR lpszValue,  
    INT nMaxLength  
);
```

The **HttpGetResponseHeader** function returns the value of the specified response header field.

Parameters

hClient

Handle to the client session.

lpszHeader

Pointer to a string which specifies the header value to be returned.

lpszValue

Pointer to a buffer which will contain the null-terminated string value of the specified header field.

nMaxLength

Maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is non-zero. If the header field does not exist or the client handle is invalid, the function returns a value of zero. To get extended error information, call **HttpGetLastError**.

Remarks

When a resource is returned by the server, it consists of three parts. The first part consists of a single line that indicates the result of the request. The second part is one or more header fields which provides specific information about the resource, such as its size in bytes. The third part consists of the resource data itself, such as the HTML document or image data. For example, this is what the response to a request for a simple HTML document can look like:

```
HTTP/1.0 200 OK  
Date: Mon, 5 Jan 2004 20:18:33 GMT  
Content-Type: text/html  
Last-Modified: Mon, 5 Jan 2004 19:34:19 GMT  
Content-Length: 115
```

```
<html>  
<head>  
<title>Simple Document</title>  
</head>  
<body>  
This is a simple HTML document.  
</body>  
</html>
```

The first line consists of the protocol version, a numeric response code and some text describing the result. The subsequent lines are the header, which is similar to the headers used in email messages. For example, the Date field specifies the date the resource was requested, the Content-

Type field specifies what type of resource was requested, and the Content-Length field specifies the size of the resource in bytes. The end of the header block is indicated by an empty line (two carriage-return/linefeed sequences), and is followed by the resource itself, in this case a simple HTML document.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetCookie](#), [HttpGetFirstCookie](#), [HttpGetFirstHeader](#), [HttpGetNextCookie](#), [HttpGetNextHeader](#), [HttpGetRequestHeader](#), [HttpSetRequestHeader](#)

HttpGetResultCode Function

```
INT WINAPI HttpGetResultCode(  
    HCLIENT hClient  
);
```

The **HttpGetResultCode** function reads the result code returned by the server in response to a command. The result code is a three-digit numeric code, and indicates if the operation succeeded, failed or requires additional action by the client.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

Result codes are three-digit numeric values returned by the server. They may be broken down into the following ranges:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttp10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpCommand](#), [HttpGetResultString](#)

HttpGetResultString Function

```
INT WINAPI HttpGetResultString(  
    HCLIENT hClient,  
    LPTSTR lpszResult,  
    INT nMaxLength  
);
```

The **HttpGetResultString** function returns the last message sent by the server along with the result code.

Parameters

hClient

Handle to the client session.

lpszResult

A pointer to the buffer that will contain the result string returned by the server.

nMaxLength

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpGetResultString** function is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **HttpGetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttp10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpCommand](#), [HttpGetResultCode](#)

HttpGetSecurityInformation Function

```
BOOL WINAPI HttpGetSecurityInformation(  
    HCLIENT hClient,  
    LPSECURITYINFO lpSecurityInfo  
);
```

The **HttpGetSecurityInformation** function returns security protocol, encryption and certificate information about the current client connection.

Parameters

hClient

Handle to the client session.

lpSecurityInfo

A pointer to a [SECURITYINFO](#) structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

This function is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value `SECURITY_PROTOCOL_NONE`.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;  
  
securityInfo.dwSize = sizeof(SECURITYINFO);  
if (HttpGetSecurityInformation(hClient, &securityInfo))  
{  
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)  
    {  
        MessageBox(NULL, _T("The connection is not secure"),  
                    _T("Connection"), MB_OK);  
    }  
    else  
    {  
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)  
        {  
            MessageBox(NULL, _T("The connection is secure"),  
                        _T("Connection"), MB_OK);  
        }  
        else  
        {  
            MessageBox(NULL, _T("The server certificate not valid"),  
                        _T("Connection"), MB_OK);  
        }  
    }  
}
```

```
}  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAsyncConnect](#), [HttpAsyncProxyConnect](#), [HttpConnect](#), [HttpDisconnect](#), [HttpGetOption](#),
[HttpProxyConnect](#), [HttpSetOption](#), [SECURITYINFO](#)

HttpGetStatus Function

```
INT WINAPI HttpGetStatus(  
    HCLIENT hClient  
);
```

The **HttpGetStatus** function returns the current status of the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpGetStatus** function returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Constant	Description
1	HTTP_STATUS_IDLE	The client is current idle and not sending or receiving data.
2	HTTP_STATUS_CONNECT	The client is establishing a connection with the server.
3	HTTP_STATUS_READ	The client is reading data from the server.
4	HTTP_STATUS_WRITE	The client is writing data to the server.
5	HTTP_STATUS_DISCONNECT	The client is disconnecting from the server.
6	HTTP_STATUS_GETDATA	The client is downloading data from the server.
7	HTTP_STATUS_PUTDATA	The client is uploading data to the server.
8	HTTP_STATUS_POSTDATA	The client is posting data to a script on the server.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session. To obtain status information about a file transfer, use the **HttpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

See Also

[HttpIsBlocking](#), [HttpIsReadable](#), [HttpIsWritable](#), [HttpGetTransferStatus](#)

HttpGetTaskError Function

```
DWORD WINAPI HttpGetTaskError(  
    UINT nTaskId  
);
```

Return the last error code for the specified asynchronous task.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has completed successfully, this function returns a value of zero. A non-zero return value indicates an error has occurred.

Remarks

The **HttpGetTaskError** function returns the last error code associated with the specified asynchronous task. If the task completed successfully, the return value will be zero. If the task is still active, the function will return the error `ST_ERROR_TASK_ACTIVE`. If the task has been suspended, the function will return `ST_ERROR_TASK_SUSPENDED`. Any other value indicates that the task completed, but the operation has failed and the error code will specify the cause of the failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshttpv10.lib`

See Also

[HttpTaskAbort](#), [HttpTaskResume](#), [HttpTaskSuspend](#), [HttpTaskWait](#)

HttpGetTaskId Function

```
UINT WINAPI HttpGetTaskId(  
    HCLIENT hClient  
);
```

Return the asynchronous task identifier associated with the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the client handle is not associated with an asynchronous task, the function will return a value of zero.

Remarks

The **HttpGetTaskId** function will return the task ID that is associated with a client session. This is a unique unsigned integer value that references the worker thread that was created to manage the asynchronous client session. This function should only be called within an event handler that is invoked by a background task that has been started using a function such as **HttpAsyncGetFile**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

See Also

[HttpTaskAbort](#), [HttpTaskDone](#), [HttpTaskResume](#), [HttpTaskSuspend](#), [HttpTaskWait](#)

HttpGetText Function

```
INT WINAPI HttpGetText(  
    HCLIENT hClient,  
    LPCTSTR lpszResource,  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);
```

The **HttpGetText** function copies the contents of a text file on the server, or the text output from a script, to the specified string buffer.

Parameters

hClient

Handle to the client session.

lpszResource

A pointer to a string that specifies the resource that will be transferred to the local system. This may be the name of a file on the server, or it may specify the name of a script that will be executed and the output returned to the caller. This string may specify a valid URL for the current server that the client is connected to.

lpszBuffer

A pointer to a string buffer which will contain the contents of the text file when the function returns. This buffer should be large enough to store the contents of the file, including a terminating null character. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. This value must be larger than zero. If this value is smaller than the actual size of the text file, the data returned will be truncated.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpGetText** function is used to download a text file or retrieve the text output from a script and store the contents in a string buffer. Because binary data can include embedded null characters which would truncate the string, this function should only be used with text files or script output that is known to be textual. For example, it is safe to use this function when a resource returns HTML or XML data, but should not be used if it returns an image or executable file.

This function has been included as a convenience for applications that need to retrieve relatively small amounts of textual data and manipulate the contents as a string. If the Unicode version of this function is called, the text is automatically converted to a Unicode string. If the maximum amount of data being returned is unknown or the amount of text is very large, it is recommended that you use the **HttpGetData** or **HttpGetFile** functions.

If you use the **HttpGetFileSize** function to determine how large the string buffer should be prior to calling this function, it is important to be aware that the actual number of characters may differ based on the end-of-line conventions used by the host operating system. For example, if you call

HttpGetFileSize to obtain the size of a text file on a UNIX system, the value will not be large enough to store the complete file because UNIX uses a single linefeed (LF) character to indicate the end-of-line, while a Windows system will use a carriage-return and linefeed (CRLF) pair. To accommodate this difference, you should always allocate extra memory for the string buffer to store the additional end-of-line characters.

HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **HttpGetTransferStatus** function.

Example

```
LPTSTR lpszBuffer = (LPTSTR)calloc(MAXFILESIZE, sizeof(TCHAR));

if (lpszBuffer == NULL)
    return;

nResult = HttpGetText(hClient, lpszRemoteFile, lpszBuffer, MAXFILESIZE);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpEnableEvents](#), [HttpGetData](#), [HttpGetFile](#), [HttpGetTransferStatus](#), [HttpPutData](#), [HttpPutFile](#), [HttpRegisterEvent](#)

HttpGetTimeout Function

```
INT WINAPI HttpGetTimeout(  
    HCLIENT hClient  
);
```

The **HttpGetTimeout** function returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

See Also

[HttpConnect](#), [HttpProxyConnect](#), [HttpSetTimeout](#)

HttpGetTransferStatus Function

```
INT WINAPI HttpGetTransferStatus(  
    HCLIENT hClient,  
    LPHTTPTRANSFERSTATUS lpStatus  
);
```

The **HttpGetTransferStatus** function returns information about the current data transfer in progress.

Parameters

hClient

Handle to the client session.

lpStatus

A pointer to an **HTTPTRANSFERSTATUS** structure which contains information about the status of the current data transfer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpGetTransferStatus** function returns information about the current data transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no data currently being transferred, this function will return the status of the last successful transfer made by the client.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

If the option HTTP_OPTION_HIRES_TIMER has been specified when connecting to the server, the values of the *dwTimeElapsed* and *dwTimeEstimated* members of the **HTTPTRANSFERSTATUS** structure will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small amounts of data over a fast network connection.

If you are uploading or downloading large files which exceed 4GB, you should use the **HttpGetTransferStatusEx** function which returns the size as a 64-bit value.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

See Also

[HttpEnableEvents](#), [HttpGetTransferStatusEx](#), [HttpRegisterEvent](#), [HTTPTRANSFERSTATUS](#), [HTTPTRANSFERSTATUSEX](#)

HttpGetTransferStatusEx Function

```
INT WINAPI HttpGetTransferStatusEx(  
    HCLIENT hClient,  
    LPHTTPTRANSFERSTATUSEX lpStatus  
);
```

The **HttpGetTransferStatusEx** function returns information about the current data transfer in progress. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpStatus

A pointer to an **HTTPTRANSFERSTATUSEX** structure which contains information about the status of the current data transfer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpGetTransferStatusEx** function returns information about the current data transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no data currently being transferred, this function will return the status of the last successful transfer made by the client.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

If the option HTTP_OPTION_HIRES_TIMER has been specified when connecting to the server, the values of the *dwTimeElapsed* and *dwTimeEstimated* members of the **HTTPTRANSFERSTATUSEX** structure will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small amounts of data over a fast network connection.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

See Also

[HttpEnableEvents](#), [HttpGetTransferStatus](#), [HttpRegisterEvent](#), [HTTPTRANSFERSTATUS](#), [HTTPTRANSFERSTATUSEX](#)

HttpInitialize Function

```
BOOL WINAPI HttpInitialize(  
    LPCTSTR lpszLicenseKey,  
    LPINITDATA lpData  
);
```

The **HttpInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpData

Pointer to an INITDATA data structure. This parameter may be NULL if the initialization data for the library is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must insure that this function is called with a valid license key.

If the *lpData* argument is specified, it must point to an INITDATA structure which has been initialized by setting the *dwSize* member to the size of the structure. All other structure members should be set to zero. If the function is successful, then the INITDATA structure will be filled with identifying information about the library.

Although it is only required that **HttpInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **HttpUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **HttpInitialize** or **HttpUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAsyncConnect](#), [HttpConnect](#), [HttpDisconnect](#), [HttpUninitialize](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

HttpIsBlocking Function

```
BOOL WINAPI HttpIsBlocking(  
    HCLIENT hClient  
);
```

The **HttpIsBlocking** function is used to determine if the client is currently performing a blocking operation.

Parameters

hClient

Handle to the client session.

Return Value

If the client is performing a blocking operation, the function returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the function returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking function may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **HttpIsBlocking** function can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshttpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpCancel](#), [HttpIsConnected](#), [HttpIsReadable](#), [HttpIsWritable](#), [HttpRead](#), [HttpWrite](#)

HttpIsConnected Function

```
BOOL WINAPI HttpIsConnected(  
    HCLIENT hClient  
);
```

The **HttpIsConnected** function is used to determine if the client is currently connected to a server.

Parameters

hClient

Handle to the client session.

Return Value

If the client is connected to a server, the function returns a non-zero value. If the client is not connected, or the client handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshttpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpIsBlocking](#), [HttpIsReadable](#), [HttpIsWritable](#)

HttpIsReadable Function

```
BOOL WINAPI HttpIsReadable(  
    HCLIENT hClient,  
    INT nTimeout,  
    LPDWORD lpdwAvail  
);
```

The **HttpIsReadable** function is used to determine if data is available to be read from the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the function returns a non-zero value. If the client cannot read any data, the function returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttp10.lib

See Also

[HttpGetStatus](#), [HttpIsBlocking](#), [HttpIsConnected](#), [HttpIsWritable](#), [HttpRead](#)

HttpIsWritable Function

```
BOOL WINAPI HttpIsWritable(  
    HCLIENT hClient,  
    INT nTimeout  
);
```

The **HttpIsWritable** function is used to determine if data can be written to the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the function returns a non-zero value. If the client cannot write any data, the function returns zero.

Remarks

Although this function can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshttpv10.lib`

See Also

[HttpGetStatus](#), [HttpIsBlocking](#), [HttpIsConnected](#), [HttpIsReadable](#), [HttpWrite](#)

HttpOpenFile Function

```
INT WINAPI HttpOpenFile(  
    HCLIENT hClient,  
    LPCTSTR lpszFileName,  
    DWORD dwOffset  
);
```

The **HttpOpenFile** function opens the specified file on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file to open.

dwOffset

Specifies a byte offset into the file. If this value is greater than zero, the server must support the ability to specify a byte range with the request to open the file, otherwise this function will fail.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

Only one file may be opened at a time for each client session. Use the **HttpCloseFile** function to close the file on the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpCloseFile](#)

HttpOpenFileEx Function

```
INT WINAPI HttpOpenFileEx(  
    HCLIENT hClient,  
    LPCTSTR lpszFileName,  
    ULARGE_INTEGER uiOffset  
);
```

The **HttpOpenFileEx** function opens the specified file on the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszFileName

Points to a string that specifies the name of the remote file to open.

uiOffset

Specifies a byte offset into the file. If this value is greater than zero, the server must support the ability to specify a byte range with the request to open the file, otherwise this function will fail.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

Only one file may be opened at a time for each client session. Use the **HttpCloseFile** function to close the file on the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpCloseFile](#)

HttpPatchData Function

```
INT WINAPI HttpPatchData(  
    HCLIENT hClient,  
    LPCTSTR lpszResource,  
    LPCTSTR lpszPatchData,  
    LPVOID lpvResult,  
    LPDWORD lpcbResult,  
    DWORD dwOptions  
);
```

The **HttpPatchData** function submits patch data to the server and returns the result in a buffer provided by the caller.

Parameters

hClient

Handle to the client session.

lpszResource

A pointer to a string that specifies the resource name that the patch data will be submitted to. Typically this is the name of a script on the server.

lpszPatchData

A pointer to a string that specifies the patch data that will be submitted to the server.

lpvResult

A pointer to a byte buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvResult* parameter. If the *lpvResult* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of data that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_PATCH_DEFAULT	The default post mode. The contents of the buffer will be submitted without encoding. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
HTTP_PATCH_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data

returned by the server.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPatchData** function is used to submit XML or JSON formatted patch data to a service, and then returns a copy of the response from the server into a local buffer. This function will not perform any encoding and will not automatically define the type of patch data being submitted. Your application is responsible for specifying the content type for the patch data, and ensuring that the XML or JSON data that is being submitted to the server is formatted correctly.

This function sends a PATCH command to the server, which is similar to a POST or PUT request. It is used to make partial updates to a resource, rather than creating or replacing it entirely. The format of the patch data is specific to the service being used. If the resource being patched does not exist, the behavior is defined by the server. If enough information is provided, it may choose to create the resource just as if a PUT command was used, or it may return an error.

Your application should use the **HttpSetRequestHeader** function to define the Content-Type header prior to calling the **HttpPatchData** function. One of the most common formats used is the JSON Merge Patch which is defined in RFC 7396. The value for the Content-Type header for this patch format is "application/merge-patch+json". Refer to your service API documentation to determine what patch formats are acceptable, along with any additional header values that must be defined.

The function may be used in one of two ways, depending on the needs of the application. The first approach is to pre-allocate a buffer large enough to store the resulting output of the script. In this case, the *lpvResult* parameter will point to the buffer that was allocated by the client and the value that the *lpcbResult* parameter points to should be initialized to the size of that buffer.

The second approach is to have the *lpvResult* parameter point to a global memory handle which will contain the data when the function returns. In this case, the value that the *lpcbResult* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

This function will cause the current thread to block until the post completes, a timeout occurs or the operation is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of the transaction while it is in progress, use the **HttpGetTransferStatus** function.

Example

```
HGLOBAL hgb1Buffer = (HGLOBAL)NULL;  
LPBYTE lpBuffer = (LPBYTE)NULL;  
DWORD cbBuffer = 0;  
  
// Store the script output into block of global memory allocated by  
// the GlobalAlloc function; the handle to this memory will be  
// returned in the hgb1Buffer parameter. Since the output from the  
// script is textual, the HTTP_PATCH_CONVERT option is used
```

```
nResult = HttpPatchData(hClient,
                        lpszResource,
                        lpszPatchData,
                        &hgblBuffer,
                        &cbBuffer,
                        HTTP_PATCH_CONVERT);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpEnableEvents](#), [HttpGetData](#), [HttpGetTransferStatus](#), [HttpPostData](#), [HttpPostJson](#), [HttpPostXml](#),
[HttpRegisterEvent](#)

HttpPostData Function

```
INT WINAPI HttpPostData(  
    HCLIENT hClient,  
    LPCTSTR lpszResource,  
    LPVOID lpvBuffer,  
    DWORD cbBuffer,  
    LPVOID lpvResult,  
    LPDWORD lpcbResult,  
    DWORD dwOptions  
);
```

The **HttpPostData** function submits the contents of the specified buffer to a script on the server and returns the result in a buffer provided by the caller.

Parameters

hClient

Handle to the client session.

lpszResource

A pointer to a string that specifies the resource that the data will be posted to on the server. Typically this is the name of a script that will be executed. This string may specify a valid URL for the current server that the client is connected to.

lpvBuffer

A pointer to the data that will be provided to the script. This parameter may be NULL if the script does not require any additional data from the client.

cbBuffer

The number of bytes to copy from the buffer. If this *lpvBuffer* parameter is NULL, this value should be zero.

lpvResult

A pointer to a byte buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvBuffer* parameter. If the *lpvResult* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of data that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_POST_DEFAULT	The default post mode. The contents of the buffer are encoded and sent as standard form data. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
HTTP_POST_CONVERT	If the data being returned from the server is textual, it is

	automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.
HTTP_POST_MULTIPART	The contents of the buffer being sent to the server consists of multipart form data and will be sent as-is without any encoding.
HTTP_POST_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail. Instead, the response is returned to the client and the function will succeed.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPostData** function is used to submit data to a script that executes on the server and then copy the output from that script into a local buffer. If you are submitting XML or JSON formatted data to the server, it is recommended that you use the **HttpPostXml** or **HttpPostJson** functions instead to ensure that the correct content type and encoding is automatically selected.

The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the resulting output of the script. In this case, the *lpvResult* parameter will point to the buffer that was allocated by the client and the value that the *lpcbResult* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvResult* parameter point to a global memory handle which will contain the data when the function returns. In this case, the value that the *lpcbResult* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

It is common for servers to return additional information about a failed request in their response to the client. If you need to process this information, use the HTTP_POST_ERRORDATA option which causes the error message to be returned in the *lpvResult* buffer provided by the caller. If this option is used, your application should call **HttpGetResultCode** to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This function will cause the current thread to block until the post completes, a timeout occurs or the operation is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of the transaction while it is in progress, use the **HttpGetTransferStatus** function.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Store the script output into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter. Since the output from the
// script is textual, the HTTP_POST_CONVERT option is used

nResult = HttpPostData(hClient,
                      lpszResource,
                      lpParameters,
                      cbParameters,
                      &hgblBuffer,
                      &cbBuffer,
                      HTTP_POST_CONVERT);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpEnableEvents](#), [HttpGetData](#), [HttpGetTransferStatus](#), [HttpPostJson](#), [HttpPostXml](#), [HttpPutData](#), [HttpRegisterEvent](#)

HttpPostFile Function

```
INT WINAPI HttpPostFile(  
    HCLIENT hClient,  
    LPCTSTR lpszFileName,  
    LPCTSTR lpszResource,  
    LPCTSTR lpszFieldName,  
    DWORD dwOptions,  
    DWORD dwReserved  
);
```

The **HttpPostFile** function posts the contents of the specified file to a script executed on the server.

Parameters

hClient

Handle to the client session.

lpszFileName

A pointer to a string that specifies the file that will be transferred from the local system. The file pathing and name conventions must be that of the local host.

lpszResource

A pointer to a string that specifies the resource on the server that the file data will be posted to. Typically this is the name of a script that is responsible for processing and storing the file data.

lpszFieldName

A pointer to a string that corresponds to the form field name that the script expects. If this parameter is NULL or an empty string, a default field name of "File1" is used.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Constant	Description
HTTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the remote file exists, it will be overwritten with the contents of the uploaded file.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPostFile** function is similar to the **HttpPutFile** function in that it can be used to upload the contents of a local file to a server. However, instead of using the PUT command, the POST command is used to send the file data to a script that is executed on the server. This method has the advantage of not requiring any special configuration settings on the server, however it does require that the script be able to process multipart/form-data as defined in RFC 2388.

To support uploading files from a form on a webpage, the FILE input type is used along with the

action that specifies the script that will accept the file data and process it. For example, the HTML code could look like this:

```
<form action="/cgi-bin/upload.cgi" method="post" enctype="multipart/form-data">  
<input type="file" name="datafile" size="20">  
<input type="submit">  
</form>
```

In this example, the script `/cgi-bin/upload.cgi` is responsible for processing the file data that is posted by the client, and the form field name "datafile" is used. The user can select a file, and when the Submit button is clicked, the file data is posted to the script. To simulate this using the **HttpPostFile** function, the *lpszFileName* parameter should be set to the name of the local file that will be posted to the server. The *lpszResource* parameter should be the name of the script, in this case `/cgi-bin/upload.cgi`. The *lpszFieldName* parameter should be specified as the string "datafile" to match the name of the field used by the form.

Note that the **HttpPostFile** function always submits the file contents as multipart/form-data with the content type set to application/octet-stream. The script that accepts the posted data must be able to parse the multipart header block and correctly process 8-bit data. If the script assumes that the data will be posted using a specific encoding type such as base64 then the file data may not be accepted or may be corrupted by the script.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the `HTTP_EVENT_PROGRESS` event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **HttpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshttpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpEnableEvents](#), [HttpGetData](#), [HttpGetFile](#), [HttpGetTransferStatus](#), [HttpPostData](#), [HttpPutData](#), [HttpPutFile](#), [HttpRegisterEvent](#)

HttpPostJson Function

```
INT WINAPI HttpPostJson(  
    HCLIENT hClient,  
    LPCTSTR lpszResource,  
    LPCTSTR lpszJsonData,  
    LPVOID lpvResult,  
    LPDWORD lpcbResult,  
    DWORD dwOptions  
);
```

The **HttpPostJson** function submits JSON formatted data to the server and returns the result in a buffer provided by the caller.

Parameters

hClient

Handle to the client session.

lpszResource

A pointer to a string that specifies the resource name that the JSON data will be submitted to. Typically this is the name of a script on the server.

lpszJsonData

A pointer to a string that specifies the JSON data that will be submitted to the server.

lpvResult

A pointer to a byte buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvResult* parameter. If the *lpvResult* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of data that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_POST_DEFAULT	The default post mode. The contents of the buffer will be submitted without encoding and should use the standard JSON format. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
HTTP_POST_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data

	returned by the server.
HTTP_POST_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail. Instead, the response is returned to the client and the function will succeed.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPostJson** function is used to submit JSON formatted data to a script that executes on the server and then copy the output from that script into a local buffer. This function automatically sets the correct content type and encoding required for submitting JSON data to a server, however it does not parse the JSON data itself to ensure that it is well-formed. Your application is responsible for ensuring that the JSON data that is being submitted to the server is formatted correctly.

The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the resulting output of the script. In this case, the *lpvResult* parameter will point to the buffer that was allocated by the client and the value that the *lpcbResult* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvResult* parameter point to a global memory handle which will contain the data when the function returns. In this case, the value that the *lpcbResult* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

If the content type for the request has not been explicitly specified, it will be automatically updated by this function to use the "application/json" content type. You can override the default content type for the request by calling the **HttpSetRequestHeader** function prior to calling this function. Most servers require you to explicitly specify what type of data is being submitted by the client and will reject requests which do not correctly identify the content type.

It is common for servers to return additional information about a failed request in their response to the client. If you need to process this information, use the HTTP_POST_ERRORDATA option which causes the error message to be returned in the *lpvResult* buffer provided by the caller. If this option is used, your application should call **HttpGetResultCode** to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This function will cause the current thread to block until the post completes, a timeout occurs or the operation is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of the transaction while it is in progress, use the **HttpGetTransferStatus** function.

Example

```
HGLOBAL hgb1Buffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
```

```

DWORD cbBuffer = 0;

// Store the script output into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter. Since the output from the
// script is textual, the HTTP_POST_CONVERT option is used

nResult = HttpPostJson(hClient,
                      lpszResource,
                      lpszJsonData,
                      &hgblBuffer,
                      &cbBuffer,
                      HTTP_POST_CONVERT);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpEnableEvents](#), [HttpGetData](#), [HttpGetTransferStatus](#), [HttpPatchData](#), [HttpPostData](#), [HttpPostXml](#), [HttpPutData](#), [HttpRegisterEvent](#)

HttpPostXml Function

```
INT WINAPI HttpPostXml(  
    HCLIENT hClient,  
    LPCTSTR lpszResource,  
    LPCTSTR lpszXmlData,  
    LPVOID lpvResult,  
    LPDWORD lpcbResult,  
    DWORD dwOptions  
);
```

The **HttpPostXml** function submits XML formatted data to the server and returns the result in a buffer provided by the caller.

Parameters

hClient

Handle to the client session.

lpszResource

A pointer to a string that specifies the resource name that the XML data will be submitted to. Typically this is the name of a script on the server.

lpszXmlData

A pointer to a string that specifies the XML data that will be submitted to the server.

lpvResult

A pointer to a byte buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvResult* parameter. If the *lpvResult* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of data that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_POST_DEFAULT	The default post mode. The contents of the buffer will be submitted without encoding and should use the standard XML format. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
HTTP_POST_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data

	returned by the server.
HTTP_POST_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail. Instead, the response is returned to the client and the function will succeed.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPostXml** function is used to submit XML formatted data to a script that executes on the server and then copy the output from that script into a local buffer. This function automatically sets the correct content type and encoding required for submitting XML data to a server, however it does not parse the XML data itself to ensure that it is well-formed. Your application is responsible for ensuring that the XML data that is being submitted to the server is formatted correctly.

The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the resulting output of the script. In this case, the *lpvResult* parameter will point to the buffer that was allocated by the client and the value that the *lpcbResult* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvResult* parameter point to a global memory handle which will contain the data when the function returns. In this case, the value that the *lpcbResult* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

If the content type for the request has not been explicitly specified, it will be automatically updated by this function to use the "text/xml" content type. You can override the default content type for the request by calling the **HttpSetRequestHeader** function prior to calling this function. Most servers require you to explicitly specify what type of data is being submitted by the client and will reject requests which do not correctly identify the content type.

It is common for servers to return additional information about a failed request in their response to the client. If you need to process this information, use the HTTP_POST_ERRORDATA option which causes the error message to be returned in the *lpvResult* buffer provided by the caller. If this option is used, your application should call **HttpGetResultCode** to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This function will cause the current thread to block until the post completes, a timeout occurs or the operation is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of the transaction while it is in progress, use the **HttpGetTransferStatus** function.

Example

```
HGLOBAL hgb1Buffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
```

```

DWORD cbBuffer = 0;

// Store the script output into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter. Since the output from the
// script is textual, the HTTP_POST_CONVERT option is used

nResult = HttpPostXml(hClient,
                    lpszResource,
                    lpszXmlData,
                    &hgblBuffer,
                    &cbBuffer,
                    HTTP_POST_CONVERT);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // output data from the script
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpEnableEvents](#), [HttpGetData](#), [HttpGetTransferStatus](#), [HttpPatchData](#), [HttpPostData](#), [HttpPostJson](#), [HttpPutData](#), [HttpRegisterEvent](#)

HttpProxyConnect Function

```
HCLIENT WINAPI HttpProxyConnect(  
    UINT nProxyType,  
    LPCTSTR LpszProxyHost,  
    UINT nProxyPort,  
    LPCTSTR LpszProxyUser,  
    LPCTSTR LpszProxyPassword,  
    LPCTSTR LpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    DWORD dwVersion,  
    LPSECURITYCREDENTIALS LpCredentials  
);
```

The **HttpProxyConnect** function is used to establish a synchronous connection with a proxy server.

Parameters

nProxyType

An unsigned integer which specifies the type of proxy that the client is connecting to. The supported proxy server types are as follows:

Constant	Description
HTTP_PROXY_NONE	A direct connection will be established with the server. When this value is specified the proxy parameters are ignored.
HTTP_PROXY_STANDARD	A standard connection is established through the specified proxy server, and all resource requests will be specified using a complete URL. This proxy type should be used with standard connections.
HTTP_PROXY_SECURE	A secure connection is established through the specified proxy server. This proxy type should be used with secure connections and the HTTP_OPTION_SECURE option should also be set via the <i>dwOptions</i> parameter.
HTTP_PROXY_WINDOWS	The configuration options for the current system should be used. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. These settings are the same proxy server settings configured in Windows.

lpszProxyHost

A pointer to a string which specifies the proxy server host name or IP address. This argument is ignored if the proxy type is set to HTTP_PROXY_NONE or HTTP_PROXY_WINDOWS and no proxy configuration has been specified for the local system.

nProxyPort

The port number that the proxy server is listening for connections on. A value of zero specifies

that the default port number 80 should be used. Note that in most cases, a proxy server is not configured to use the default port. This argument is ignored if the proxy type is set to HTTP_PROXY_NONE or HTTP_PROXY_WINDOWS and no proxy configuration has been specified for the local system.

lpzProxyUser

A pointer to a string which specifies the user name that will be used to authenticate the client session to the proxy server. If the server does not require user authentication, then a NULL pointer may be passed in this argument.

lpzProxyPassword

A pointer to a string which specifies the password that will be used to authenticate the client session to the proxy server. If the server does not require user authentication, then a NULL pointer may be passed in this argument.

lpzRemoteHost

A pointer to a string which specifies the name of the server to connect to through the proxy server. This may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number 80 should be used. For standard connections, the default port number is 80. For secure connections, the default port number is 443.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested. If the server does not support the keep-alive option, the client will automatically reconnect when each resource is requested. Although it will not provide any performance benefits, this allows the option to be used with all servers.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from

	the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_ERRORDATA	This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.
HTTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
HTTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
HTTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

dwVersion

The requested protocol version used when sending requests to the server. The high word should specify the major version, and the low word should specify the minor version number. The HTTPVERSION macro can be used to create version value.

lpCredentials

Pointer to credentials structure [SECURITYCREDENTIALS](#). This parameter is only used if the HTTP_OPTION_SECURE option is specified for the connection. This parameter may be NULL, in

which case no client credentials will be provided to the server. If client credentials are required, the fields *dwSize*, *lpszCertStore*, and *lpszCertName* must be defined, while other fields may be left undefined. Set *dwSize* to the size of the **SECURITYCREDENTIALS** structure.

Return Value

If the function succeeds, the return value is a handle to the client session. If the function fails, the return value is **INVALID_CLIENT**. To get extended error information, call **HttpGetLastError**.

Remarks

If the **HTTP_PROXY_WINDOWS** proxy type is specified, then the proxy configuration for the local system is used. If no proxy server has been defined, then the proxy-related parameters will be ignored and the function will establish a connection directly to the server.

The username and password information is only used when connecting to a server which supports version 1.1 or later of the protocol.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **HttpAttachThread** function.

Specifying the **HTTP_OPTION_FREETHREAD** option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshttpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpConnect](#), [HttpDisconnect](#), [HttpInitialize](#)

HttpPutData Function

```
INT WINAPI HttpPutData(  
    HCLIENT hClient,  
    LPCTSTR lpszRemoteFile,  
    LPVOID lpvBuffer,  
    DWORD dwLength,  
    DWORD dwReserved  
);
```

The **HttpPutData** function copies the contents of the specified buffer and stores it in a file on the server.

Parameters

hClient

Handle to the client session.

lpszRemoteFile

A pointer to a string that specifies the file on the server that will be created. This string may specify a valid URL for the current server that the client is connected to.

lpvBuffer

A pointer to a buffer which will contain the data to be transferred from the client and stored in a file on the server.

dwLength

The number of bytes to copy from the buffer.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPutData** function is used to store the contents of the specified buffer in a file on the server. Not all servers permit files to be created using this method, and some may require that specific configuration changes be made to the server in order to support this functionality. Consult your server's technical reference documentation to see if it supports the PUT command, and if so, what must be done to enable it. It may be required that the client authenticate itself using the **HttpAuthenticate** function prior to uploading the data.

This function is designed for uploading the contents of a buffer to be stored as a file on the server. If you need to use the PUT command to as part of a RESTful API where the PUT command is used to submit data to the server, use the **HttpPutDataEx** function. That function will return the success or failure responses back to the client.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a transfer while it is in progress, use the

HttpGetTransferStatus function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpEnableEvents](#), [HttpGetData](#), [HttpGetFile](#), [HttpGetTransferStatus](#), [HttpPostData](#), [HttpPutDataEx](#), [HttpPutFile](#), [HttpRegisterEvent](#)

HttpPutDataEx Function

```
INT WINAPI HttpPutDataEx(  
    HCLIENT hClient,  
    LPCTSTR lpszResource,  
    LPVOID lpvBuffer,  
    DWORD cbBuffer,  
    LPVOID lpvResult,  
    LPDWORD lpcbResult,  
    DWORD dwOptions  
);
```

The **HttpPutDataEx** function submits the contents of the specified buffer to the server using the PUT command and returns the server's response to the command.

Parameters

hClient

Handle to the client session.

lpszResource

A pointer to a string which specifies the resource path on the server. This string may specify a valid URL for the current server that the client is connected to.

lpvBuffer

A pointer to a buffer which will contain the data to be transferred from the client and submitted to the server. This parameter may not be NULL.

cbBuffer

The number of bytes to copy from the buffer. This value must be larger than zero.

lpvResult

A pointer to a byte buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvResult* parameter. If the *lpvResult* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of data that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_PUT_DEFAULT	The default mode. The contents of the buffer are sent unmodified and the data returned by the server is copied to the result buffer exactly as it is returned from the server. This option should be used if the server is expected to return binary data in the response to the PUT request.
HTTP_PUT_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character

	sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the data being submitted to the server, only on the data returned by the server.
HTTP_PUT_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail. Instead, the response is returned to the client and the function will succeed.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPutDataEx** function is used to submit data to the server using the PUT command. This is typically used with RESTful APIs and is similar in functionality to the POST command. However, the data that is submitted will be sent to the server as-is and will not be encoded.

The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the response from the server. In this case, the *lpvResult* parameter will point to the buffer allocated by the client and the value that the *lpcbResult* parameter points to should be initialized to the size of that buffer.

The second method is to have the *lpvResult* parameter point to a global memory handle which will reference the data when the function returns. In this case, the value that the *lpcbResult* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur.

If the Unicode version of this function is called and the server returns a text response to the PUT request, it is the application's responsibility to convert the contents of the *lpvResult* buffer to UTF-16 text. If the server returns text, most likely it will use UTF-8 encoding.

If the content type for the request has not been explicitly specified, it will be automatically updated by this function to use the "application/octet-stream" content type. You can override the default content type for the request by calling the **HttpSetRequestHeader** function prior to calling this function. Most servers require you to explicitly specify what type of data is being submitted by the client and will reject requests which do not correctly identify the content type.

It is common for servers to return additional information about a failed request in their response to the client. If you need to process this information, use the HTTP_PUT_ERRORDATA option which causes the error message to be returned in the *lpvResult* buffer provided by the caller. If this option is used, your application should call **HttpGetResultCode** to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This function will cause the current thread to block until the data transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a data transfer while it is in progress, use the

`HttpGetTransferStatus` function.

Example

```
HGLOBAL hgb1Response = (HGLOBAL)NULL;
LPBYTE lpResponse = (LPBYTE)NULL;
DWORD cbResponse = 0;

// Store the response into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgb1Buffer parameter. Since the response from
// the server is textual, the HTTP_PUT_CONVERT option is used

nResult = HttpPutDataEx(hClient,
                        lpszResource,
                        lpRequest,
                        cbRequest,
                        &hgb1Response,
                        &cbResponse,
                        HTTP_PUT_CONVERT);

if (nResult != HTTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // response from the server
    lpResponse = (LPBYTE)GlobalLock(hgb1Response);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgb1Response);
    GlobalFree(hgb1Response);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshttpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpEnableEvents](#), [HttpGetData](#), [HttpGetFile](#), [HttpGetTransferStatus](#), [HttpPostData](#), [HttpPutData](#), [HttpRegisterEvent](#)

HttpPutFile Function

```
INT WINAPI HttpPutFile(  
    HCLIENT hClient,  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszRemoteFile,  
    DWORD dwOptions  
    DWORD dwOffset  
);
```

The **HttpPutFile** function transfers the specified file on the local system to the server.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A pointer to a string that specifies the file that will be transferred from the local system. The file pathing and name conventions must be that of the local system.

lpszRemoteFile

A pointer to a string that specifies the file on the server that will be created, overwritten or appended to. This string may specify a valid URL for the current server that the client is connected to.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Constant	Description
HTTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the remote file exists, it will be overwritten with the contents of the uploaded file.

dwOffset

Specifies a byte offset into the file. If this value is greater than zero, the server must support the ability to specify a byte range with the request to open the file, otherwise this function will fail.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPutFile** function is used to transfer a file from the local system to a server. Not all servers permit files to be uploaded and some may require that specific configuration changes be made to the server in order to support this functionality. Consult your server's technical reference documentation to see if it supports the PUT command, and if so, what must be done to enable it. It may be required that the client authenticate itself using the **HttpAuthenticate** function prior to uploading the file.

To upload large files that are over 4GB, use the **HttpPutFileEx** function.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be

periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **HttpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpEnableEvents](#), [HttpGetFile](#), [HttpGetTransferStatus](#), [HttpPostData](#), [HttpPostFile](#), [HttpPutData](#), [HttpPutFileEx](#), [HttpRegisterEvent](#)

HttpPutFileEx Function

```
INT WINAPI HttpPutFileEx(  
    HCLIENT hClient,  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszRemoteFile,  
    DWORD dwOptions  
    ULARGE_INTEGER uiOffset  
);
```

The **HttpPutFileEx** function transfers the specified file on the local system to the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A pointer to a string that specifies the file that will be transferred from the local system. The file pathing and name conventions must be that of the local host.

lpszRemoteFile

A pointer to a string that specifies the file on the server that will be created, overwritten or appended to. The file naming conventions must be that of the host operating system.

dwOptions

An unsigned integer that specifies one or more options. This parameter may be any one of the following values:

Constant	Description
HTTP_TRANSFER_DEFAULT	This option specifies the default transfer mode should be used. If the remote file exists, it will be overwritten with the contents of the uploaded file.

uiOffset

Specifies a byte offset into the file. If this value is greater than zero, the server must support the ability to specify a byte range with the request to open the file, otherwise this function will fail.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPutFileEx** function is used to transfer a file from the local system to a server. Not all servers permit files to be uploaded and some may require that specific configuration changes be made to the server in order to support this functionality. Consult your server's technical reference documentation to see if it supports the PUT command, and if so, what must be done to enable it. It may be required that the client authenticate itself using the **HttpAuthenticate** function prior to uploading the file.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification

must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **HttpGetTransferStatusEx** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpEnableEvents](#), [HttpGetFileEx](#), [HttpGetTransferStatusEx](#), [HttpPostData](#), [HttpPostFile](#), [HttpPutData](#), [HttpRegisterEvent](#)

HttpPutText Function

```
INT WINAPI HttpPutText(  
    HCLIENT hClient,  
    LPCTSTR lpszRemoteFile,  
    LPCTSTR lpszBuffer  
);
```

The **HttpPutText** function creates a text file on the server using the contents of a string buffer.

Parameters

hClient

Handle to the client session.

lpszRemoteFile

A pointer to a string that specifies the text file on the server that will be created or overwritten. This string may specify a valid URL for the current server that the client is connected to.

lpszBuffer

A pointer to a string that contains the text that will be stored in the file.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPutText** function is used to create a text file on the server from the contents of a string. If the specified file already exists on the server, its contents will be overwritten. Not all servers permit files to be created using this method, and some may require that specific configuration changes be made to the server in order to support this functionality. Consult your server's technical reference documentation to see if it supports the PUT command, and if so, what must be done to enable it. It may be required that the client authenticate itself using the **HttpAuthenticate** command prior to uploading the data.

If the Unicode version of this function is called, the string will be automatically converted to UTF-8 and then uploaded to the server. If you wish to upload UTF-16 Unicode or ANSI text to the server, you must use the **HttpPutData** function. This function should never be used to upload binary data.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a file transfer while it is in progress, use the **HttpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpEnableEvents](#), [HttpGetText](#), [HttpGetTransferStatus](#), [HttpPutData](#), [HttpPutFile](#),
[HttpRegisterEvent](#)

HttpPutTextEx Function

```
INT WINAPI HttpPutTextEx(  
    HCLIENT hClient,  
    LPCTSTR lpszResource,  
    LPCTSTR lpszBuffer,  
    LPVOID lpvResult,  
    LPDWORD lpcbResult,  
    DWORD dwOptions  
);
```

The **HttpPutTextEx** function submits the contents of a string buffer to the server using the PUT command and returns the server's response to the command.

Parameters

hClient

Handle to the client session.

lpszResource

A pointer to a string which specifies the resource path on the server. This string may specify a valid URL for the current server that the client is connected to.

lpszBuffer

A pointer to a string that contains the text that will be submitted to the server. This parameter cannot be NULL and may not specify an empty string.

lpvResult

A pointer to a byte buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvResult* parameter. If the *lpvResult* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of data that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_PUT_DEFAULT	The default mode. automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the data being submitted to the server, only on the data returned by the server.
HTTP_PUT_ERRORDATA	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the function to fail.

Instead, the response is returned to the client and the function will succeed.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpPutTextEx** function is used to submit the contents of a null terminated string to the server using the PUT command. This is typically used with RESTful APIs and is similar in functionality to the POST command. However, the text that is submitted will not be encoded in the same way that the **HttpPostData** function submits data.

If the Unicode version of this function is called, the text will be converted to UTF-8 and then submitted to the server. If you wish to submit the contents of the string as UTF-16 Unicode text, or the payload contains binary data, you must use the **HttpPutDataEx** function. This function should never be used to upload binary data. If the server returns a text response and the Unicode version of this function is called, it is the application's responsibility to convert the contents of the *lpvResult* buffer to UTF-16 text. If the server returns text, most likely it will use UTF-8 encoding.

If the content type for the request has not been explicitly specified, it will be automatically updated by this function to use the "text/plain" content type. You can override the default content type for the request by calling the **HttpSetRequestHeader** function prior to calling this function. Most servers require you to explicitly specify what type of data is being submitted by the client and will reject requests which do not correctly identify the content type.

It is common for servers to return additional information about a failed request in their response to the client. If you need to process this information, use the HTTP_PUT_ERRORDATA option which causes the error message to be returned in the *lpvResult* buffer provided by the caller. If this option is used, your application should call **HttpGetResultCode** to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This function will cause the current thread to block until the data transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a data transfer while it is in progress, use the **HttpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttp10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpEnableEvents](#), [HttpGetText](#), [HttpGetTransferStatus](#), [HttpPutDataEx](#), [HttpPutFile](#), [HttpPutText](#), [HttpRegisterEvent](#)

HttpRead Function

```
INT WINAPI HttpRead(  
    HCLIENT hClient,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **HttpRead** function reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hClient

Handle to the client session.

lpBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

When **HttpRead** is called and the client is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

See Also

[HttpIsBlocking](#), [HttpIsReadable](#), [HttpIsWritable](#), [HttpWrite](#)

HttpRegisterEvent Function

```
INT WINAPI HttpRegisterEvent(  
    HCLIENT hClient,  
    UINT nEventId,  
    HTTPEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

The **HttpRegisterEvent** function registers an event handler for the specified event.

Parameters

hClient

Handle to the client session.

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Constant	Description
HTTP_EVENT_CONNECT	The connection to the server has completed.
HTTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
HTTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
HTTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
HTTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
HTTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
HTTP_EVENT_PROGRESS	The client is in the process of sending or receiving a file on the data channel. This event is called periodically during a transfer so that the client can update any user interface components such as

	a status control or progress bar.
HTTP_EVENT_REDIRECT	This event is generated when a the server indicates that the requested resource has been moved to a new location. The new resource location may be on the same server, or it may be located on another server. Check the value of the Location header field to determine where the resource has been moved to.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Remarks

The **HttpRegisterEvent** function associates a callback function with a specific event. The event handler is an **HttpEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This function is typically used to register an event handler that is invoked while a file is being uploaded or downloaded. The HTTP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one HTTP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpDisableEvents](#), [HttpEnableEvents](#), [HttpEventProc](#), [HttpFreezeEvents](#), [HttpGetTransferStatus](#)

HttpSetBearerToken Function

```
INT WINAPI HttpSetBearerToken(  
    HCLIENT hClient,  
    LPTSTR lpszBearerToken  
);
```

The **HttpSetBearerToken** function sets the OAuth 2.0 bearer token used to authenticate the client session with a web service.

Parameters

hClient

Handle to the client session.

lpszBearerToken

A pointer to a null terminated string buffer which contains the bearer token used to authorize client requests. If this parameter is NULL or a zero length string, the current bearer token will be cleared and no client authentication will be performed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

Using OAuth 2.0 requires you to understand the process of how to request the bearer (access) token. Obtaining a bearer token requires registering your application with the web service provider, getting a unique client ID associated with your application and then requesting the token using the appropriate scope for the service. Obtaining the initial token will typically involve interactive confirmation on the part of the user, requiring they grant permission to your application to access the service.

Your application should not store a bearer token for later use. They have a relatively short lifespan, typically about an hour, and are designed to be used with that session. You should specify offline access as part of the OAuth 2.0 scope if necessary and store the refresh token provided by the service. The refresh token has a much longer validity period and can be used to obtain a new bearer token when needed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAuthenticate](#), [HttpGetBearerToken](#), [HttpRequestHeader](#)

HttpSetCookie Function

```
BOOL WINAPI HttpSetCookie(  
    HCLIENT hClient,  
    LPCTSTR lpszCookieName,  
    LPCTSTR lpszCookieValue  
);
```

The **HttpSetCookie** function sends the specified cookie to the server when a resource is requested.

Parameters

hClient

Handle to the client session.

lpszCookieName

Pointer to a string which specifies the name of the cookie that will be sent to the server when the next resource is requested.

lpszCookieValue

Pointer to a string which specifies the value of the cookie. To delete a cookie that has been previously set, this parameter should be NULL or point to an empty string.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpSetCookie** function submits the cookie name and value to the server when a resource is requested or data is posted to a script. For more information about cookies and how they are used, refer to the **HttpGetCookie** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

See Also

[HttpGetCookie](#), [HttpGetFirstCookie](#), [HttpGetNextCookie](#), [HttpSetRequestHeader](#)

HttpSetEncodingType Function

```
INT WINAPI HttpSetEncodingType(  
    HCLIENT hClient  
    INT nEncodingType  
);
```

The **HttpSetEncodingType** function specifies the type of encoding to be applied to the content of a HTTP request.

Parameters

hClient

Handle to the client session.

nEncodingType

Specifies the content encoding type; currently-supported values are:

Constant	Description
HTTP_ENCODING_NONE	No encoding will be applied to the content of a request, and no Content-Type header will be generated.
HTTP_ENCODING_URL	URL encoding will be applied to the content of a request, and a Content-Type header will be generated with the value "application/x-www-form-urlencoded"
HTTP_ENCODING_XML	URL encoding will be applied to the content of a request, EXCEPT that spaces will not be replaced by '+'. This encoding type is intended for use with XML parsers that do not recognize '+' as a space. A Content-Type header will be generated with the value "application/x-www-form-urlencoded".

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpSetEncodingType** function explicitly sets the type of encoding used when optional parameter data is submitted with a request for a resource. By default, data is URL encoded and the content type will be designated as application/x-www-form-urlencoded.

If an application specifies HTTP_ENCODING_NONE, then the parameter data is not encoded and there is no content type header created by default. The client application can create its own Content-Type header field by calling the **HttpRequestHeader** function if necessary.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

HttpSetFormProperties Function

```
INT WINAPI HttpSetFormProperties(  
    HFORM hForm,  
    LPHTTPFORMPROPERTIES LpFormProp  
);
```

The **HttpSetFormProperties** function updates the properties of the specified form.

Parameters

hForm

Handle to the virtual form.

LpFormProp

Points to a HTTPFORMPROPERTIES structure which specifies the new properties for the form.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpCreateForm](#), [HttpGetFormProperties](#), [HttpSubmitForm](#), [HTTPFORMPROPERTIES](#)

HttpSetLastError Function

```
VOID WINAPI HttpSetLastError(  
    DWORD dwErrorCode  
);
```

The **HttpSetLastError** function sets the last error code for the current thread. This function is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or HTTP_ERROR. Those functions which call **HttpSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **HttpGetLastError** function. The use of **HttpGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

See Also

[HttpGetErrorString](#), [HttpGetLastError](#)

HttpSetOption Function

```
INT WINAPI HttpSetOption(  
    HCLIENT hClient,  
    DWORD dwOption,  
    BOOL bEnabled  
);
```

The **HttpSetOption** function enables or disables the specified option.

Parameters

hClient

Handle to the client session.

dwOption

An unsigned integer which specifies one of the following options:

Constant	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_KEEPALIVE	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested. If the server does not support the keep-alive option, the client will automatically reconnect when each resource is requested. Although it will not provide any performance benefits, this allows the option to be used with all servers.
HTTP_OPTION_REDIRECT	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
HTTP_OPTION_NOUSERAGENT	This option specifies the client should not include a User-Agent header with any requests made during the session. The user agent is a string which is used to identify the client application to the server.
HTTP_OPTION_ERRORDATA	This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with HTTP_OPTION_REDIRECT.
HTTP_OPTION_FREETHREAD	This option specifies the handle returned by this

	function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.
HTTP_OPTION_HIRES_TIMER	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data over fast network connections.

bEnabled

An integer value which determines if the option should be enabled or disabled. A non-zero value specifies that the option should be enabled, while a zero value specifies that the option should be disabled.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpSetOption** function can only enable or disable the options listed above. Other options are only available when the connection is initially established. For example, you cannot use this function to enable security options after the connection has been made.

If you use HTTP_OPTION_FREETHREAD to permit any thread to reference a client handle allocated in another thread, your application is responsible for ensuring it does not attempt to submit requests using the same handle on different threads at the same time. If two different threads attempt to perform an operation using the same handle, there is no guarantee as to which thread will complete the operation first.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttp10.lib

See Also

[HttpAsyncConnect](#), [HttpAsyncProxyConnect](#), [HttpConnect](#), [HttpGetOption](#), [HttpProxyConnect](#)

HttpSetPriority Function

```
INT WINAPI HttpSetPriority(  
    HCLIENT hClient,  
    INT nPriority  
);
```

The **HttpSetPriority** function specifies the priority for file transfers.

Parameters

hClient

Handle to the client session.

nPriority

An integer value which specifies the new priority for file transfers. It may be one of the following values:

Constant	Description
HTTP_PRIORITY_NORMAL	The default priority which balances resource utilization and transfer speed. It is recommended that most applications use this priority.
HTTP_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the transfer. It is typically used with worker threads running in the background when the amount of time required perform the transfer is not critical.
HTTP_PRIORITY_LOW	This priority lowers the overall resource utilization for the transfer and meters the bandwidth allocated for the transfer. This priority will increase the average amount of time required to complete a file transfer.
HTTP_PRIORITY_HIGH	This priority increases the overall resource utilization for the transfer, allocating more memory for internal buffering. It can be used when it is important to transfer the file quickly, and there are no other threads currently performing file transfers at the time.
HTTP_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization while attempting to transfer the file as quickly as possible. If the file transfer is being performed in the main UI thread, this priority can cause the application to appear to become non-responsive. No events will be generated during the transfer.

Return Value

If the function succeeds, the return value is the previous file transfer priority. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpSetPriority** function can be used to control the processor usage, memory and network

bandwidth allocated for file transfers. The default priority balances resource utilization and transfer speed while ensuring that a single-threaded application remains responsive to the user. Lower priorities reduce the overall resource utilization at the expense of transfer speed. For example, if you create a worker thread to download a file in the background and want to ensure that it has a minimal impact on the process, the `HTTP_PRIORITY_BACKGROUND` value can be used.

Higher priority values increase the memory allocated for the transfers and increases processor utilization for the transfer. The `HTTP_PRIORITY_CRITICAL` priority maximizes transfer speed at the expense of system resources. It is not recommended that you increase the file transfer priority unless you understand the implications of doing so and have thoroughly tested your application. If the file transfer is being performed in the main UI thread, increasing the priority may interfere with the normal processing of Windows messages and cause the application to appear to become non-responsive. It is also important to note that when the priority is set to `HTTP_PRIORITY_CRITICAL`, normal progress events will not be generated during the transfer.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshttpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetPriority](#)

HttpSetRequestHeader Function

```
BOOL WINAPI HttpSetRequestHeader(  
    HCLIENT hClient,  
    LPCTSTR lpszHeader,  
    LPCTSTR lpszValue  
);
```

The **HttpSetRequestHeader** function sets the value of the specified request header field.

Parameters

hClient

Handle to the client session.

lpszHeader

Points to a string which specifies the request header field name.

lpszValue

Points to a string which specifies the value of the request header field.

Return Value

If the function succeeds, the return value is non-zero. If the client handle is invalid, the function returns a value of zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpSetRequestHeader** function defines a header field and value that is submitted to the server when a resource is requested. If the header field has already been defined, it will be replaced by the new value. There are a number of header fields which are automatically created by the library, and others that are conditionally created depending on whether or not certain options are specified. The following header fields are automatically created by the library:

Header Field	Description
Accept	This header specifies the types of resources that are acceptable to the client. By default, all resource data types are accepted by the client.
Authorization	This header is automatically created if the client uses the <code>HttpAuthenticate</code> function to authenticate a client session.
Connection	This header determines if the connection is maintained after a resource has been requested, or if the connection should be immediately closed. The value of this header depends on whether the <code>HTTP_OPTION_KEEPALIVE</code> option has been specified.
Content-Length	This header defines the length of the data that is being posted to the server or the size of a file being uploaded to the server.
Content-Type	This header defines the content type for data posted to the server. This header is created if URL encoding is specified. The specific type of encoding used can be set by calling the <code>HttpSetEncodingType</code> function.
Host	This header specifies the name of the server that the client has

	connected to. This is automatically generated when any resource is requested.
Pragma	This header is used to control caching performed by the server. If the option HTTP_OPTION_NOCACHE has been specified, this header will automatically be defined with the value "no-cache".
Proxy-Authorization	This header is automatically created if a proxy connection has been established and a username and password is required to authenticate the client session.

Request headers are generated by functions that send resource requests. In some cases, header values are supplied by the requesting function only if the application has not previously defined the header. For others, the requesting function overrides what the application may have defined.

If you use this function to set the Authorization header to a custom value for this client session, you must not call the **HttpAuthenticate** function. The **HttpAuthenticate** function will always override any custom Authorization header value and replace it with the credentials token generated from the username and password provided.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAuthenticate](#), [HttpGetFirstHeader](#), [HttpGetNextHeader](#), [HttpGetRequestHeader](#), [HttpGetResponseHeader](#), [HttpSetEncodingType](#)

HttpSetTimeout Function

```
INT WINAPI HttpSetTimeout(  
    HCLIENT hClient,  
    INT nTimeout  
);
```

The **HttpSetTimeout** function sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

See Also

[HttpGetTimeout](#), [HttpIsBlocking](#), [HttpIsReadable](#), [HttpIsWritable](#)

HttpSubmitForm Function

```
INT WINAPI HttpSubmitForm(  
    HCLIENT hClient,  
    HFORM hForm,  
    LPVOID lpvResult,  
    LPDWORD lpcbResult,  
    DWORD dwOptions  
);
```

The **HttpSubmitForm** function submits the contents of the specified form to a script on the server and returns the result in a buffer provided by the caller.

Parameters

hClient

Handle to the client session.

hForm

Handle to the virtual form which contains the data to be submitted to the server.

lpvResult

A pointer to a byte buffer which will contain the data returned by the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpcbResult

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvBuffer* parameter. If the *lpvResult* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of data that was returned.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_SUBMIT_DEFAULT	The default post mode. The contents of the buffer are encoded and sent as standard form data. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
HTTP_SUBMIT_CONVERT	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpSubmitForm** function is used to submit form data to a script that executes on the server and then copy the output from that script into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the resulting output of the script. In this case, the *lpvResult* parameter will point to the buffer that was allocated by the client and the value that the *lpcbResult* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvResult* parameter point to a global memory handle which will contain the data when the function returns. In this case, the value that the *lpcbResult* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

This function will cause the current thread to block until the post completes, a timeout occurs or the operation is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **HttpEnableEvents**, or by registering a callback function using the **HttpRegisterEvent** function.

To determine the current status of a transaction while it is in progress, use the **HttpGetTransferStatus** function.

Example

```
HFORM hForm = INVALID_FORM;
HGLOBAL hgblResult = (HGLOBAL)NULL;
DWORD cbResult = 0;
INT nResult = 0;

hForm = HttpCreateForm(_T("/login.php"), HTTP_METHOD_POST, HTTP_FORM_ENCODED);

if (hForm == INVALID_FORM)
    return;

HttpAddFormField(hForm, _T("UserName"), lpszUserName, (DWORD)-1L, 0);
HttpAddFormField(hForm, _T("Password"), lpszPassword, (DWORD)-1L, 0);

nResult = HttpSubmitForm(hClient, hForm, &hgblResult, &cbResult, 0);
HttpDestroyForm(hForm);

if (hgblResult != NULL)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblResult);

    // lpBuffer points to data returned by the server after the form
    // data was submitted

    GlobalUnlock(hgblResult);
    GlobalFree(hgblResult);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAddFormField](#), [HttpAddFormFile](#), [HttpClearForm](#), [HttpCreateForm](#), [HttpDeleteFormField](#),
[HttpRegisterEvent](#)

HttpTaskAbort Function

```
BOOL WINAPI HttpTaskAbort(  
    UINT nTaskId,  
    DWORD dwMilliseconds  
);
```

Abort the specified asynchronous task.

Parameters

nTaskId

The task identifier.

dwMilliseconds

An unsigned integer that specifies the number of milliseconds to wait for the background task to abort.

Return Value

If the function succeeds and the worker thread has terminated, the return value is non-zero. A return value of zero indicates that the worker thread is still running or an error has occurred. To get extended error information, call the **HttpGetLastError** function.

Remarks

The **HttpTaskAbort** function signals the background worker thread associated with the task ID to abort the current operation and terminate as soon as possible. If the *dwMilliseconds* parameter has a value of zero, the function returns immediately after the background thread has been signaled. If the *dwMilliseconds* parameter is non-zero, the function will wait that amount of time for the background thread to terminate.

This function should never be called from within the event handler for an asynchronous task because it can cause the process to deadlock. To abort a file transfer within an event handler, use the **HttpCancel** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttp10.lib

See Also

[HttpTaskDone](#), [HttpTaskResume](#), [HttpTaskSuspend](#), [HttpTaskWait](#)

HttpTaskDone Function

```
BOOL WINAPI HttpTaskDone(  
    UINT nTaskId  
);
```

Determine if an asynchronous task has completed.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has completed, this function returns a non-zero value. A return value of zero indicates that the worker thread is still running or an error has occurred. To get extended error information, call the **HttpGetLastError** function.

Remarks

The **HttpTaskDone** function is used to determine if the specified asynchronous task has completed. If you use this function to poll the status of a background task from within the main UI thread, you must ensure that Windows messages are processed so that the application remains responsive to the end-user. To check if a background transfer has completed, it is recommended that you use a timer to periodically call this function rather than calling it repeatedly within a loop.

To determine if the task completed successfully, the **HttpGetTaskError** function will return the last error code associated with the task. A return value of zero indicates success, while a non-zero return value specifies an error code that indicates the cause of the failure. The last error code for the task can also be retrieved using the **HttpTaskWait** function, which causes the application to wait for the asynchronous task to complete.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

See Also

[HttpGetTaskError](#), [HttpTaskAbort](#), [HttpTaskResume](#), [HttpTaskSuspend](#), [HttpTaskWait](#)

HttpTaskResume Function

```
BOOL WINAPI HttpTaskResume(  
    UINT nTaskId  
);
```

Resume execution of an asynchronous task.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has resumed, this function returns a non-zero value. A return value of zero indicates that an error has occurred. To get extended error information, call the **HttpGetLastError** function.

Remarks

The **HttpTaskResume** function resumes execution of the background worker thread that was previously suspended using the **HttpTaskSuspend** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

See Also

[HttpTaskAbort](#), [HttpTaskDone](#), [HttpTaskSuspend](#), [HttpTaskWait](#)

HttpTaskSuspend Function

```
BOOL WINAPI HttpTaskSuspend(  
    UINT nTaskId  
);
```

Suspend execution of an asynchronous task.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has resumed, this function returns a non-zero value. A return value of zero indicates that an error has occurred. To get extended error information, call the **HttpGetLastError** function.

Remarks

The **HttpTaskSuspend** function will suspend execution of the background worker thread associated with the task. Once the task has been suspended, it will no longer be scheduled for execution, however the client session will remain active and the task may be resumed using the **HttpTaskResume** function. Note that if a task is suspended for a long period of time, the background operation may fail because it has exceeded the timeout period imposed by the server.

This function should never be called from within the event handler for an asynchronous task because it can cause the process to deadlock.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

See Also

[HttpTaskAbort](#), [HttpTaskDone](#), [HttpTaskResume](#), [HttpTaskWait](#)

HttpTaskWait Function

```
BOOL WINAPI HttpTaskWait(  
    UINT nTaskId,  
    DWORD dwMilliseconds,  
    DWORD dwReserved,  
    LPDWORD lpdwElapsed,  
    LPDWORD lpdwError  
);
```

Wait for an asynchronous task to complete.

Parameters

nTaskId

The task identifier.

dwMilliseconds

An unsigned integer that specifies the number of milliseconds to wait for the background task to complete.

dwReserved

An unsigned integer reserved for future use. This value should always be zero.

lpdwElapsed

A pointer to an unsigned integer that will contain elapsed time in milliseconds when the function returns. If this information is not required, this parameter may be NULL.

lpdwError

A pointer to an unsigned integer that will contain the error code associated with the completed task. If this information is not required, this parameter may be NULL.

Return Value

If the function succeeds and the worker thread has terminated, the return value is non-zero. A return value of zero indicates that the worker thread is still running or an error has occurred. To get extended error information, call the **HttpGetLastError** function.

Remarks

The **HttpTaskWait** function waits for the specified task to complete. If the task is active and the *dwMilliseconds* parameter is non-zero, this function will cause the current working thread to block until the task completes or the amount of time exceeds the number of milliseconds specified by the caller. If the *dwMilliseconds* parameter is zero, then this function will poll the status of the task and return immediately to the caller.

If the specified task has already completed at the time this function is called, the function will return immediately without causing the current thread to block. If the *lpdwElapsed* parameter is not NULL, it will contain the number of milliseconds that it took for the task to complete. If the *lpdwError* parameter is not NULL, it will contain the last error code value that was set by the worker thread before it terminated. If this value is zero, that means that the background operation was successful and no error occurred. A non-zero value will indicate that the background operation has failed.

You should not call this function from the main UI thread with a long timeout period to wait for a background task to complete. Windows messages will not be processed while this function is blocked waiting for the background task to complete, and this can cause your application to

appear non-responsive to the end-user. If you have a GUI application and you need to periodically check to see if a task has completed, create a timer to periodically call the **HttpTaskDone** function. When it returns a non-zero value (indicating that the task has completed), you can safely call **HttpTaskWait** to obtain the elapsed time and last error code without blocking the current thread.

This function should never be called from within the event handler for an asynchronous task because it can cause the process to deadlock.

Example

```
UINT nTaskId;

// Begin a file transfer in the background

nTaskId = HttpAsyncGetFile(hClient,
                          lpszLocalFile,
                          lpszRemoteFile,
                          HTTP_TRANSFER_DEFAULT,
                          0,
                          NULL,
                          0);

if (nTaskId != 0)
{
    DWORD dwError = NO_ERROR;
    DWORD dwElapsed = 0;

    // Wait for the transfer to complete
    HttpTaskWait(nTaskId, INFINITE, &dwElapsed, &dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: cshttpv10.lib

See Also

[HttpTaskDone](#), [HttpTaskResume](#), [HttpTaskSuspend](#), [HttpTaskWait](#)

HttpUninitialize Function

```
VOID WINAPI HttpUninitialize();
```

The **HttpUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **HttpInitialize** call before it can call any of the other library functions. When it has completed the use of library, the application must call **HttpUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There must be a call to **HttpUninitialize** for every successful call to **HttpInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshttpv10.lib

See Also

[HttpDisconnect](#), [HttpInitialize](#)

HttpUploadFile Function

```
BOOL WINAPI HttpUploadFile(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszFileURL,  
    UINT nTimeout  
    DWORD dwOptions  
    LPHTTPTRANSFERSTATUS lpStatus  
    HTTPEVENTPROC lpEventProc  
    DWORD_PTR dwParam  
);
```

The **HttpUploadFile** function uploads the specified file from the local system to the server.

Parameters

lpszLocalFile

A pointer to a string that specifies the file on the local system that will be uploaded to the server. The file pathing and name conventions must be that of the local host.

lpszFileURL

A pointer to a string that specifies the complete URL of the file that will be created or overwritten on the server. The URL must follow the conventions for the Hypertext Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed using the system Control Panel.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.

HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.

lpStatus

A pointer to an HTTPTRANSFERSTATUS structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpUploadFile** function provides a convenient way for an application to upload a file in a single function call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session and then upload the file to the server. The URL must be complete, and specify either a standard or secure HTTP scheme:

```
[http|https]://[username : password] @] remotehost [:remoteport] / [path / ...] [filename]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "http" scheme is used and the HTTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, the "https" scheme must be specified.

Not all web servers permit files to be uploaded and some may require that specific configuration changes be made to the server in order to support this functionality. Consult your server's technical reference documentation to see if it supports the PUT command, and if so, what must be done to enable it. It may be required that the URL specify a username and password to upload a file.

The *lpStatus* parameter can be used by the application to determine the final status of the transfer, including the total number of bytes copied, the amount of time elapsed and other information related to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The *lpEventProc* parameter specifies a pointer to a function which will be periodically called during the file transfer process. This can be used to check the status of the transfer by calling

HttpGetTransferStatus and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The *dwParam* parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the *this* pointer as the value, and then cast it back to an object pointer inside the callback function. If no event handler is required, then a NULL pointer can be specified as the value for *lpEventProc* and the *dwParam* parameter will be ignored.

The **HttpUploadFile** function is designed to provide a simpler interface for uploading a file. However, complex connections such as those using a specific proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **HttpConnect** or **HttpProxyConnect** and then use **HttpPutFile** to upload the file. If the server does not support the PUT command, you may be able to upload files using the **HttpPostFile** function. Refer to that function for more information.

Example

```
HTTPTRANSFERSTATUS httpStatus;
LPCTSTR lpszLocalFile = _T("c:\\temp\\database.mdb");
LPCTSTR lpszFileURL =
_T("http://update:secret@www.example.com/updates/database.mdb");
BOOL bResult;

// Upload the file using the specified URL
bResult = HttpUploadFile(lpszLocalFile,
                        lpszFileURL,
                        HTTP_OPTION_PASSIVE,
                        &httpStatus,
                        NULL, 0);

if (!bResult)
{
    TCHAR szError[128];

    // Display a message box that describes the error
    HttpGetErrorString(HttpGetLastError(), szError, 128);
    MessageBox(NULL, szError, NULL, MB_ICONEXCLAMATION|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpEventProc](#), [HttpDownloadFile](#), [HttpGetTransferStatus](#), [HttpPostFile](#), [HttpPutFile](#), [HTTPTRANSFERSTATUS](#)

HttpUploadFileEx Function

```
BOOL WINAPI HttpUploadFileEx(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszFileURL,  
    UINT nTimeout  
    DWORD dwOptions  
    LPHTTPTRANSFERSTATUSEX lpStatus  
    HTTPEVENTPROC lpEventProc  
    DWORD_PTR dwParam  
);
```

The **HttpUploadFileEx** function uploads the specified file from the local system to the server. This version of the function is designed to support files that are larger than 4GB.

Parameters

lpszLocalFile

A pointer to a string that specifies the file on the local system that will be uploaded to the server. The file pathing and name conventions must be that of the local host.

lpszFileURL

A pointer to a string that specifies the complete URL of the file that will be created or overwritten on the server. The URL must follow the conventions for the Hypertext Transfer Protocol and may specify either a standard or secure connection, alternate port number, username, password and optional working directory.

nTimeout

The number of seconds that the client will wait for a response before failing the operation. A value of zero specifies that the default timeout period of sixty seconds will be used.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
HTTP_OPTION_NOCACHE	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
HTTP_OPTION_PROXY	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed using the system Control Panel.
HTTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is

	established.
HTTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
HTTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.

lpStatus

A pointer to an HTTPTRANSFERSTATUSEX structure which contains information about the status of the current file transfer. If this information is not required, a NULL pointer may be specified as the parameter.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **HttpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpUploadFileEx** function provides a convenient way for an application to upload a file in a single function call. Based on the connection information specified in the URL, it will connect to the server, authenticate the session and then upload the file to the server. The URL must be complete, and specify either a standard or secure HTTP scheme:

```
[http|https]://[username : password] @ remotehost [:remoteport] / [path / ...] [filename]
```

If no user name and password is provided, then the client session will be authenticated as an anonymous user. The URL scheme will always determine if the connection is secure, not the option. In other words, if the "http" scheme is used and the HTTP_OPTION_SECURE option is specified, that option will be ignored. To establish a secure connection, the "https" scheme must be specified.

Not all web servers permit files to be uploaded and some may require that specific configuration changes be made to the server in order to support this functionality. Consult your server's technical reference documentation to see if it supports the PUT command, and if so, what must be done to enable it. It may be required that the URL specify a username and password to upload a file.

The *lpStatus* parameter can be used by the application to determine the final status of the transfer, including the total number of bytes copied, the amount of time elapsed and other information related to the transfer process. If this information isn't needed, then this parameter may be specified as NULL.

The *lpEventProc* parameter specifies a pointer to a function which will be periodically called during

the file transfer process. This can be used to check the status of the transfer by calling **HttpGetTransferStatus** and then update the program's user interface. For example, the callback function could calculate the percentage for how much of the file has been transferred and then update a progress bar control. The *dwParam* parameter is used in conjunction with the event handler and specifies a user-defined value that is passed to the callback function. One common use in a C++ program is to pass the *this* pointer as the value, and then cast it back to an object pointer inside the callback function. If no event handler is required, then a NULL pointer can be specified as the value for *lpEventProc* and the *dwParam* parameter will be ignored.

The **HttpUploadFileEx** function is designed to provide a simpler interface for uploading a file. However, complex connections such as those using a specific proxy server or a secure connection which uses a client certificate will require the program to establish the connection using **HttpConnect** or **HttpProxyConnect** and then use **HttpPutFileEx** to upload the file. If the server does not support the PUT command, you may be able to upload files using the **HttpPostFile** function. Refer to that function for more information.

Example

```
HTTPTRANSFERSTATUSEX httpStatus;
LPCTSTR lpszLocalFile = _T("c:\\temp\\database.mdb");
LPCTSTR lpszFileURL =
_T("http://update:secret@www.example.com/updates/database.mdb");
BOOL bResult;

// Upload the file using the specified URL
bResult = HttpUploadFileEx(lpszLocalFile,
                           lpszFileURL,
                           HTTP_OPTION_PASSIVE,
                           &httpStatus,
                           NULL, 0);

if (!bResult)
{
    TCHAR szError[128];

    // Display a message box that describes the error
    HttpGetErrorString(HttpGetLastError(), szError, 128);
    MessageBox(NULL, szError, NULL, MB_ICONEXCLAMATION|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpEventProc](#), [HttpDownloadFileEx](#), [HttpGetTransferStatusEx](#), [HttpPostFile](#), [HttpPutFileEx](#), [HTTPTRANSFERSTATUSEX](#)

HttpValidateUrl Function

```
BOOL WINAPI HttpValidateUrl(  
    LPCTSTR lpszURL  
);
```

The **HttpValidateUrl** function determines if a string represents a valid HTTP URL.

Parameters

lpszURL

A pointer to a string that specifies the URL to validate.

Return Value

If the specified URL is valid and the host name can be resolved to an IP address, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpValidateUrl** function will check the value of a string to ensure that it represents a complete, valid URL using either a standard or secure HTTP scheme. This function will not establish a connection with the server to verify that it exists, it will only attempt to resolve the host name to an IP address. If the remote host is specified as an IP address, this function will check to make sure that the address is formatted correctly. Note that if you wish to specify an IPv6 address, you must enclose the address in brackets.

To establish a connection with a server using a URL, use the **HttpConnectUrl** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpConnectUrl](#)

HttpVerifyFile Function

```
BOOL WINAPI HttpVerifyFile(  
    HCLIENT hClient,  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszRemoteFile,  
    DWORD dwReserved  
);
```

The **HttpVerifyFile** function attempts to verify that the size of a file on the local system is the same as the specified file on the server.

Parameters

hClient

Handle to the client session.

lpszLocalFile

A pointer to a string that specifies the name of file on the local system.

lpszRemoteFile

A pointer to a string that specifies the name of the file on the server.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetLastError**.

Remarks

The **HttpVerifyFile** function will attempt to verify that the contents of the local and remote files are identical by comparing the size of the files. This function is provided for compatibility with the File Transfer Protocol API, and should not be considered a reliable method for comparing files. Web servers may not consistently return file size information for dynamically created content such as HTML pages which use server-side includes.

It is not recommended that you use this function with text files because of the different end-of-line conventions used by different operating systems. For example, a text file on a Windows system uses a carriage-return and linefeed pair to indicate the end of a line of text. However, on a UNIX system, a single linefeed is used to indicate the end of a line. This can cause the **HttpVerifyFile** function to indicate the files are not identical, even though the only difference is in the end-of-line characters that are used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpDeleteFile](#), [HttpGetFile](#), [HttpGetFileSize](#), [HttpPutFile](#)

HttpWrite Function

```
INT WINAPI HttpWrite(  
    HCLIENT hClient,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **HttpWrite** function sends the specified number of bytes to the server.

Parameters

hClient

Handle to the client session.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for an asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshttp10.lib

See Also

[HttpIsBlocking](#), [HttpIsReadable](#), [HttpIsWritable](#), [HttpRead](#)

Hypertext Transfer Protocol Data Structures

- HTTPFORMPROPERTIES
- HTTPTRANSFERSTATUS
- HTTPTRANSFERSTATUSEX
- INITDATA
- SECURITYCREDENTIALS
- SECURITYINFO
- SYSTEMTIME

HTTPFORMPROPERTIES Structure

This structure is used by the **HttpGetFormProperties** and **HttpSetFormProperties** function to return and modify the properties of the specified form.

```
typedef struct _HTTPFORMPROPERTIES
{
    UINT    nFormMethod;
    UINT    nFormType;
    LPCTSTR lpszFormAction;
    DWORD   dwReserved1;
    DWORD   dwReserved2;
} HTTPFORMPROPERTIES, *LPHTTPFORMPROPERTIES;
```

Members

nFormMethod

An unsigned integer value which specifies how the form data will be submitted to the server. It may be one of the following values:

Constant	Description
HTTP_METHOD_GET	The form data should be submitted using the GET command. This method should be used when the amount of form data is relatively small. If the total amount of form data exceeds 2048 bytes, it is recommended that the POST method be used instead.
HTTP_METHOD_POST	The form data should be submitted using the POST command. This is the preferred method of submitting larger amounts of form data. If the total amount of form data exceeds 2048 bytes, it is recommended that the POST method be used.

nFormType

An unsigned integer value which specifies the type of form and how the data will be encoded when it is submitted to the server. It may be one of the following values:

Constant	Description
HTTP_FORM_ENCODED	The form data should be submitted as URL encoded values. This is typically used when the GET method is used to submit the data to the server.
HTTP_FORM_MULTIPART	The form data should be submitted as multipart form data. This is typically used when the POST method is used to submit a file to the server. Note that the script must understand how to process multipart form data if this form type is specified.

lpszFormAction

A pointer to a string which specifies the name of the resource that the form data will be submitted to. Typically this is the name of a script that is executed on the server.

dwReserved1

A reserved structure member.

dwReserved2

A reserved structure member.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpCreateForm](#), [HttpGetFormProperties](#), [HttpSetFormProperties](#), [HttpSubmitForm](#)

HTTPTRANSFERSTATUS Structure

This structure is used by the **HttpGetTransferStatus** function to return information about a data transfer in progress.

```
typedef struct _HTTPTRANSFERSTATUS
{
    DWORD    dwBytesTotal;
    DWORD    dwBytesCopied;
    DWORD    dwBytesPerSecond;
    DWORD    dwTimeElapsed;
    DWORD    dwTimeEstimated;
} HTTPTRANSFERSTATUS, *LPHTTPTRANSFERSTATUS;
```

Members

dwBytesTotal

The total number of bytes that will be transferred. If the data is being downloaded from the server to the local host, this is the size of the requested resource. If the data is being uploaded from the local host to the server, it is the size of the local file or memory buffer. If the size of the resource cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the data transfer is completed. This is based on the average number of bytes transferred per second.

Remarks

If the option `HTTP_OPTION_HIRES_TIMER` has been specified when connecting to the server, the values returned in the *dwTimeElapsed* and *dwTimeEstimated* members will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small amounts of data over a fast network connection.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

See Also

[HttpEnableEvents](#), [HttpGetTransferStatus](#), [HttpGetTransferStatusEx](#), [HttpRegisterEvent](#), [HTTPTRANSFERSTATUSEX](#)

HTTPTRANSFERSTATUSEX Structure

This structure is used by the **HttpGetTransferStatusEx** function to return information about a data transfer in progress. This structure is designed for use with extended functions that support files larger than 4GB.

```
typedef struct _HTTPTRANSFERSTATUSEX
{
    ULARGE_INTEGER uiBytesTotal;
    ULARGE_INTEGER uiBytesCopied;
    DWORD          dwBytesPerSecond;
    DWORD          dwTimeElapsed;
    DWORD          dwTimeEstimated;
} HTTPTRANSFERSTATUSEX, *LPHTTPTRANSFERSTATUSEX;
```

Members

uiBytesTotal

The total number of bytes that will be transferred. If the data is being downloaded from the server to the local host, this is the size of the requested resource. If the data is being uploaded from the local host to the server, it is the size of the local file or memory buffer. If the size of the resource cannot be determined, this value will be zero.

uiBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the data transfer is completed. This is based on the average number of bytes transferred per second.

Remarks

If the option `HTTP_OPTION_HIRES_TIMER` has been specified when connecting to the server, the values returned in the *dwTimeElapsed* and *dwTimeEstimated* members will be in milliseconds instead of seconds. You can use this option to obtain more accurate elapsed times when uploading or downloading small amounts of data over a fast network connection.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

See Also

[HttpEnableEvents](#), [HttpGetTransferStatus](#), [HttpGetTransferStatusEx](#), [HttpRegisterEvent](#), [HTTPTRANSFERSTATUS](#)

INITDATA Structure

This structure contains information about the SocketTools library when the initialization function returns.

```
typedef struct _INITDATA
{
    DWORD        dwSize;
    DWORD        dwVersionMajor;
    DWORD        dwVersionMinor;
    DWORD        dwVersionBuild;
    DWORD        dwOptions;
    DWORD_PTR    dwReserved1;
    DWORD_PTR    dwReserved2;
    TCHAR        szDescription[128];
} INITDATA, *LPINITDATA;
```

Members

dwSize

Size of this structure. This structure member must be set to the size of the structure prior to calling the initialization function. Failure to do so will cause the function to fail.

dwVersionMajor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the major version number for the library.

dwVersionMinor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the minor version number for the library.

dwVersionBuild

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the build number for the library.

dwOptions

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved1

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved2

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

szDescription

A null terminated string which describes the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

SECURITYCREDENTIALS Structure

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on

	what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that

will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A pointer to a null terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A pointer to a null terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A pointer to a null terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A pointer to a null terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
------------	-------------

CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
----	---

MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
----	--

ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.
------	--

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

A pointer to a null terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used for SSH connections

and should always be NULL when establishing a secure connection using SSL or TLS.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* member should be set to CREDENTIAL_STORE_FILENAME and the *lpzCertStore* member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS#12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the *lpzPassword* member or the library will be unable to access the certificate.

Note that the *lpzUserName* and *lpzPassword* members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

The TLS 1.1 and TLS 1.2 protocols are only supported on Windows 7, Windows Server 2008 R2 and later versions of the platform. If these options are specified and the application is running on Windows XP or Windows Vista, the protocol version will be downgraded to TLS 1.0 for backwards compatibility with those platforms.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

SECURITYINFO Structure

This structure contains information about a secure connection that has been established.

```
typedef struct _SECURITYINFO
{
    DWORD        dwSize;
    DWORD        dwProtocol;
    DWORD        dwCipher;
    DWORD        dwCipherStrength;
    DWORD        dwHash;
    DWORD        dwHashStrength;
    DWORD        dwKeyExchange;
    DWORD        dwCertStatus;
    SYSTEMTIME   stCertIssued;
    SYSTEMTIME   stCertExpires;
    LPCTSTR      lpszCertIssuer;
    LPCTSTR      lpszCertSubject;
    LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to **sizeof(SECURITYINFO)** prior to passing the address of this structure to the function. Note that if this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the protocol that was selected to establish the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The

	<p>correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.</p>
SECURITY_PROTOCOL_TLS10	<p>The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS11	<p>The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS12	<p>The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.</p>

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_CIPHER_RC2	The RC2 block cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
SECURITY_CIPHER_DES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher, using 56-bit

	keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher (also known as the Rijndael cipher) is a fixed block size cipher which use a key size of 128, 192 or 256 bits. This cipher is supported on Windows XP SP3 and later versions of the operating system.
SECURITY_CIPHER_SKIPJACK	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
SECURITY_CIPHER_BLOWFISH	The Blowfish block cipher was selected. This is a variable key length cipher up to 448 bits, using a 64-bit block size.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected. One of the following values will be returned:

Constant	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected. One of the following values will be returned:

--	--

Constant	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography. This key exchange algorithm is only supported on Windows XP SP3 and later versions of the operating system.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server. This member only has meaning for connections using the SSL or TLS protocols. One of the following values will be returned:

Constant	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertIssuer

A pointer to a string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertSubject

A pointer to a string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszFingerprint

A pointer to a string which contains a sequence of hexadecimal values that uniquely identify the server. This member is only used when a connection has been established using the Secure Shell (SSH) protocol.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Unicode: Implemented as Unicode and ANSI versions.

SYSTEMTIME Structure

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include windows.h.

Hypertext Transfer Protocol Server Library

Implements a server that enables the application to access documents using the Hypertext Transfer Protocol.

Reference

- [Functions](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

File Name	CSHTSV10.DLL
Version	10.0.1468.2518
LibID	39B2930E-60DC-42D6-B388-6D6CD768DBE7
Import Library	CSHTSV10.LIB
Dependencies	None
Standards	RFC 1945, RFC 2616, RFC 3875

Overview

This library provides an interface for implementing an embedded, lightweight server that can be used to provide access to documents and other resources using the Hypertext Transfer Protocol. The server can accept connections from any standard web browser, third-party applications or programs developed using the SocketTools HTTP client API.

The application specifies an initial server configuration and then responds to events that are generated when the client sends a request to the server. An application may implement only minimal handlers for most events, in which case the default actions are performed for most standard HTTP commands. However, an application may also use the event mechanism to filter specific commands or to extend the protocol by providing custom implementations of existing commands or add entirely new commands.

The server includes support for CGI scripting, virtual hosting, client authentication and the creation of virtual directories and files. The server also supports secure connections using SSL/TLS. Secure connections require that a valid SSL certificate be installed on the system.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This library provides an implementation of a multithreaded server which should only be used with

languages that support the creation of multithreaded applications. It is important that you do not link against static libraries which were not built with support for threading.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Hypertext Transfer Protocol Server Functions

Function	Description
HttpAddVirtualHost	Add a new virtual host to the server virtual host table
HttpAddVirtualHostAlias	Add an alternate host name for an existing virtual host
HttpAddVirtualPath	Add a new virtual path for the specified host
HttpAddVirtualUser	Add a new virtual user for the specified host
HttpAuthenticateClient	Authenticate the client and assign access rights for the session
HttpCheckVirtualPath	Determine if the client has permission to access the specified virtual path
HttpCreateServerCredentials	Create a new server security credentials structure
HttpDeleteAllClientHeaders	Delete all of the request or response headers for the specified client session
HttpDeleteClientHeader	Delete a request or response header for the specified client session
HttpDeleteServerCredentials	Delete a previously created security credentials structure
HttpDeleteVirtualHost	Delete a virtual host associated with the specified server
HttpDeleteVirtualPath	Delete a virtual path from the specified virtual host
HttpDeleteVirtualUser	Delete a virtual user from the specified virtual host
HttpDisconnectClient	Disconnect the specific client session, closing the control channel and aborting any file transfer
HttpEnableClientAccess	Enable or disable access rights for the specified client session
HttpEnableCommand	Enable or disable a specific server command
HttpEnumServerClients	Returns a list of active client sessions established with the specified server
HttpGetActiveClient	Return the client ID for the active client session associated with the current thread
HttpGetAllClientHeaders	Return all client request or response headers in a string buffer
HttpGetClientAccess	Return the access rights that have been granted to the client session
HttpGetClientAddress	Return the IP address of the specified client session
HttpGetClientCredentials	Return the credentials for the specified client session
HttpGetClientDirectory	Return the root document directory for a client session
HttpGetClientHeader	Return the value of a request or response header for the specified client session
HttpGetClientIdleTime	Return the idle timeout period for the specified client
HttpGetClientLocalPath	Return the full local path for the specified virtual path
HttpGetClientServer	Return the handle to the server that created the specified client session
HttpGetClientThreadId	Returns the thread ID associated with the specified client session
HttpGetClientUserName	Return the user name associated with the specified client session

HttpGetClientVariable	Return the value of a CGI environment variable for the specified client
HttpGetClientVirtualHost	Return the name of the virtual host the client used to establish the connection
HttpGetClientVirtualHostId	Return the virtual host ID associated with the specified client session
HttpGetClientVirtualPath	Return the virtual path for a local file on the server
HttpGetCommandFile	Return the full path to the local file name or directory specified by the client
HttpGetCommandLine	Return the complete command line issued by the client
HttpGetCommandName	Return the name of the command that was issued by the client
HttpGetCommandQuery	Return the query parameters included with the command
HttpGetCommandResource	Return the path for the resource requested by the client
HttpGetCommandResult	Return the result code and a description of the last command processed by the server
HttpGetCommandUrl	Return the complete URL of the resource requested by the client
HttpGetProgramExitCode	Return the exit code of the last program executed by the client
HttpGetProgramName	Return the name of the CGI program executed by the client
HttpGetProgramOutput	Return a copy of the standard output from a CGI program executed by the client
HttpGetProgramText	Return a copy of the standard output from a CGI program in a string buffer
HttpGetServerAddress	Return the IP address for the server
HttpGetServerDirectory	Return the full path to the root directory assigned to the specified server
HttpGetServerError	Return information about the last server error that occurred
HttpGetServerIdentity	Return the identity and version information for the specified server
HttpGetServerLogFile	Return the current log file format and full path for the file
HttpGetServerMemoryUsage	Return the amount of memory allocated for the server and all client sessions
HttpGetServerName	Return the host name assigned to the server or specified client session
HttpGetServerOptions	Return the configuration options for the specified server
HttpGetServerPriority	Return the current priority assigned to the specified server
HttpGetServerStackSize	Return the initial size of the stack allocated for threads created by the server
HttpGetServerTransferInfo	Return information about the current file transfer
HttpGetServerUuid	Return the UUID assigned to the specified server
HttpGetServerUuidString	Return the UUID assigned to the server as a printable string
HttpGetVirtualHostId	Return the virtual host ID associated with the specified hostname
HttpGetVirtualHostName	Return the hostname associated with the specified virtual host ID

HttpIsClientAuthenticated	Determine if the specified client session has been authenticated
HttpIsCommandEnabled	Determine if the specified command is currently enabled or disabled
HttpReceiveRequest	Receive the request that was sent by the client to the server
HttpRedirectRequest	Redirect the request from the client to another URL
HttpRegisterHandler	Register a CGI program for use and associate it with a file name extension
HttpRegisterProgram	Register a CGI program for use and associate it with a virtual path on the server
HttpRenameServerLogFile	Rename or delete the current log file being updated by the server
HttpRequireAuthentication	Send a response to the client indicating that authentication is required
HttpSendErrorResponse	Send a customized error response to the specified client
HttpSendResponse	Send a response from the server to the specified client
HttpSendResponseData	Send additional data to the client in response to a command
HttpServerAsyncNotify	Enable or disable asynchronous notification of changes in server status
HttpServerDisableTrace	Disable logging of network function calls
HttpServerEnableTrace	Enable logging of network function calls to a file
HttpServerInitialize	Initialize the library and validate the specified license key at runtime
HttpServerProc	Callback function used to process server events
HttpServerRestart	Restart the server, terminating all active client sessions
HttpServerResume	Resume accepting client connections on the specified server
HttpServerStart	Start the server and begin accepting client connections
HttpServerStop	Stop the server and terminate all active client connections
HttpServerSuspend	Suspend accepting client connections on the specified server
HttpServerThrottle	Limit the number of active client connections, connections per address and connection rate
HttpServerUninitialize	Terminate use of the library by the application
HttpSetClientAccess	Change the access rights associated with the specified client session
HttpSetClientHeader	Create or change the value of a request or response header for the client session
HttpSetClientIdleTime	Change the idle timeout period for the specified client session
HttpSetClientVariable	Create or change the value of a CGI environment variable for the specified client
HttpSetCommandFile	Change the name of the local file or directory that is the target of the current command
HttpSetServerError	Set the last error code for the specified server session
HttpSetServerIdentity	Change the identity and version information for the specified server
HttpSetServerLogFile	Change the current log format, level of detail and file name

HttpSetServerName	Change the hostname assigned to the specified server or client session
HttpSetServerPriority	Change the priority assigned to the specified server
HttpSetServerStackSize	Change the initial size of the stack allocated for threads created by the server.

HttpAddVirtualHost Function

```
UINT WINAPI HttpAddVirtualHost(  
    HSERVER hServer,  
    LPCTSTR lpszHostName,  
    UINT nHostPort,  
    LPCTSTR lpszDirectory  
);
```

Add a new virtual host to the server virtual host table.

Parameters

hServer

The server handle.

lpszHostName

A pointer to a string which specifies the hostname that will be added to the virtual host table. This parameter must specify a valid hostname and cannot be a NULL pointer or a zero-length string.

nHostPort

An integer value which specifies the port number for the virtual host. This value must be zero or the same value as the original port number that the server was configured to use.

lpszDirectory

A pointer to a NULL terminated string which specifies the root document directory for the virtual host. If this parameter is NULL or a zero-length string, the virtual host will use the same root directory that was specified when the server was started. This parameter may contain environment variables enclosed in % symbols.

Return Value

If the function is successful, it will return a non-zero integer value that identifies the virtual host. If the function fails, it will return `INVALID_VIRTUAL_HOST` and the last error code will be updated to indicate the cause of the failure.

Remarks

Virtual hosting is a method for sharing multiple domain names on a single instance of a server. The client provides the server with the hostname that it has used to establish the connection, and that name is compared against a table of virtual hosts configured for the server. If the hostname matches a virtual host, the client will use the root directory and any virtual paths that have been assigned to that host.

When the server is first started, a default virtual host with an ID of zero is automatically created and is identified as `VIRTUAL_HOST_DEFAULT`. This virtual host uses the same hostname, port number and root directory that the server instance was created with. The application should treat all other host IDs as opaque values and never make assumptions about how they are allocated.

The *nHostPort* parameter should always be specified with a value of zero, or the same port number that the server was configured to use. Port-based virtual hosting is currently not supported and this parameter is included for future use.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAddVirtualHostAlias](#), [HttpAddVirtualPath](#), [HttpAddVirtualUser](#), [HttpDeleteVirtualHost](#)

HttpAddVirtualHostAlias Function

```
BOOL WINAPI HttpAddVirtualHostAlias(  
    HSERVER hServer,  
    UINT nHostId,  
    LPCTSTR lpszHostAlias  
);
```

Add an alternate host name for an existing virtual host.

Parameters

hServer

The server handle.

nHostId

An integer value which identifies the virtual host.

lpszHostAlias

A pointer to a string which specifies the alias for the virtual host. The alias must be a valid domain name that uniquely identifies the host. This parameter cannot be a NULL pointer or specify an empty string.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid or the virtual host ID does not specify a valid host, the function will return zero. If the function fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **HttpAddVirtualHostAlias** function adds an alias for an existing virtual host. This enables a client to establish a connection using a number of different domain names which all reference the same virtual host. When the server responds to the client, it will identify itself with the primary domain name assigned to the virtual host rather than the alias provided by the client.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshtsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAddVirtualHost](#), [HttpAddVirtualPath](#)

HttpAddVirtualPath Function

```
BOOL WINAPI HttpAddVirtualPath(  
    HSERVER hServer,  
    UINT nHostId,  
    DWORD dwFileAccess,  
    LPCTSTR lpszVirtualPath,  
    LPCTSTR lpszLocalPath  
);
```

Add a new virtual path for the specified host.

Parameters

hServer

The server handle.

nHostId

An integer value which identifies the virtual host.

dwFileAccess

An integer value which specifies the access clients will be given to the virtual path. For a list of file access permissions, see [User and File Access Constants](#).

lpszVirtualPath

A pointer to a string which specifies the virtual path that will be created. This parameter cannot be a NULL pointer or an empty string. The maximum length of the virtual path is 1024 characters.

lpszLocalPath

A pointer to a string which specifies the local directory or file name that the virtual path will be mapped to. This path must exist and can be no longer than MAX_PATH characters. This parameter cannot be a NULL pointer or an empty string.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid or the virtual host ID does not specify a valid host, the function will return zero. If the function fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **HttpAddVirtualPath** function maps a virtual path name to a directory or file name on the local system. Virtual paths are assigned to specific hosts and if multiple virtual hosts are created for the server, each can have its own virtual paths which map to different files. To create a virtual path for the default server, the *nHostId* parameter should be specified as VIRTUAL_HOST_DEFAULT.

It is recommended that the *lpszLocalPath* parameter always specify the full path to the local file or directory. If the path is relative, it will be considered relative to the current working directory for the process and expanded to its full path name. The local path can include environment variables surrounded by % symbols. For example, if the value %ProgramData% is included in the path, it will be expanded to the full path for the common application data folder. The local path cannot specify a Windows system folder or the root directory of a mounted drive volume.

The local file or directory does not need to be located in the document root directory for the server or virtual host. It can specify any valid local path that the server process has the appropriate permissions to access. You should exercise caution when creating virtual paths to files or

directories outside of the server root directory. If the *lpszLocalPath* parameter specifies a directory, clients will have access to that directory and all subdirectories using its virtual path.

If you wish to password protect the virtual file or directory, include the HTTP_ACCESS_PROTECTED flag in the file permissions. The default command handlers will recognize this flag and require that the client authenticate itself to grant access to the resource. If the server application implements a custom command handler, it is responsible for checking for the presence of this flag and perform the appropriate checks to ensure that the client session has been authenticated.

If the server was started in restricted mode, the client will be unable to access documents outside of the server root directory and its subdirectories. This restriction also applies to virtual paths that reference documents or other resources outside of the root directory. To allow a client to access a document outside of the server root directory, the **HttpSetClientAccess** function should be used to grant the client HTTP_ACCESS_READ permission.

The **HttpGetClientVirtualPath** function will return the virtual path that is associated with a local file or directory. The **HttpGetClientLocalPath** function will return the full path to a local file or directory that is mapped to a virtual path.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAddVirtualHost](#), [HttpCheckVirtualPath](#), [HttpGetClientLocalPath](#), [HttpGetClientVirtualPath](#), [HttpDeleteVirtualPath](#)

HttpAddVirtualUser Function

```
BOOL WINAPI HttpAddVirtualUser(  
    HSERVER hServer,  
    UINT nHostId,  
    DWORD dwUserAccess,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszDirectory  
);
```

Add a new virtual user for the specified host.

Parameters

hServer

The server handle.

nHostId

An integer value which identifies the virtual host.

dwUserAccess

An integer value which specifies the access clients will be given when authenticated as this user. For a list of user access permissions, see [User and File Access Constants](#).

lpszUserName

A pointer to a string which specifies the user name. The maximum length of a username is 63 characters and it is recommended that names be limited to alphanumeric characters. Whitespace, control characters and certain symbols such as path delimiters and wildcard characters are not permitted. If an invalid character is included in the name, the function will fail with an error indicating the username is invalid. This parameter cannot be NULL and the name must be at least three characters in length. Usernames are not case sensitive.

lpszPassword

A pointer to a string which specifies the user password. The maximum length of a password is 63 characters and is limited to printable characters. Whitespace and control characters are not permitted. If an invalid character is included in the password, the function will fail with an error indicating the password is invalid. This parameter cannot be NULL and must be at least one character in length. Passwords are case sensitive.

lpszDirectory

A pointer to a string which specifies the local directory that is considered to be the virtual user's home directory. This path must exist and can be no longer than MAX_PATH characters. The maximum length of the local path is 260 characters. The directory cannot be located in a Windows system folder or the root directory of a mounted disk volume. If this parameter is NULL or an empty string, the server root directory will be assigned as the user home directory.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid or the virtual host ID does not specify a valid host, the function will return zero. If the function fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **HttpAddVirtualUser** function adds a virtual user that is associated with the specified virtual

host. If a client attempts to access a protected document and provides credentials, the server will attempt to automatically authenticate the session by searching for virtual user with the same username and password. If a match is found, then the client session is assigned the same access permissions as the virtual user.

If the server is started with the HTTP_SERVER_MULTUSER option, then documents in the virtual user's home directory can be accessed by specifying their username using a specially formatted request URL. For example, if a virtual user named "Thomas" is created, the documents in that user's home directory could be accessed as `http://servername/~thomas/document.html`

All files and subdirectories in the user's home directory are considered to be read-only. A client cannot create files in a user's home directory, even if they are authenticated as that user. In addition, CGI programs and scripts cannot be executed from a user's home directory.

If you wish to modify the information for a user, it is not necessary to delete the username first. If this function is called with a username that already exists, that record is replaced with the values passed to this function.

The virtual users created by this function exist only as long as the server is active. If you wish to maintain a persistent database of users and passwords, you are responsible for its implementation based on the requirements of your specific application. For example, a simple implementation would be to store the user information in a local XML or INI file and then read that configuration file after the server has started, calling this function for each user that is listed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshtsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAddVirtualHost](#), [HttpDeleteVirtualUser](#)

HttpAuthenticateClient Function

```
BOOL WINAPI HttpAuthenticateClient(  
    HSERVER hServer,  
    UINT nClientId,  
    DWORD dwUserAccess  
);
```

Authenticate the client and assign access rights for the session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies one or more user access rights. For a list of user access rights that can be granted to the client, see [User and File Access Constants](#).

Return Value

If the the client session could be authenticated, the return value is non-zero. If the server handle and client ID do not specify a valid client session, or the client has already been authenticated, this function will return zero.

Remarks

The **HttpAuthenticateClient** function is used to authenticate a specific client session, typically in response to an HTTP_CLIENT_USERAUTH event that indicates a client has provided authentication credentials as part of the request for a document or other resource.

To enable the server to automatically authenticate a client session, use the **HttpAddVirtualUser** function to add one or more virtual users. The server will search the list of virtual users for a match to the credentials provided by the client and will set the appropriate permissions for the session without requiring a event handler to manually authenticate the session using this function.

If the server was started with the HTTP_SERVER_LOCALUSER option and the client session is not authenticated using this function, the server will attempt to authenticate the client session using the local Windows user database. Although this option can be convenient because it does not require the implementation of an event handler for the HTTP_CLIENT_USERAUTH event, it can be used by clients to attempt to discover valid usernames and passwords for the local system. It is recommended that you use the **HttpAddVirtualUser** function to create virtual users rather than using the local user database.

It is recommended that most applications specify HTTP_ACCESS_DEFAULT as the *dwUserAccess* value for a client session, since this allows the server automatically grant the appropriate access based on the server configuration options.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAddVirtualUser](#), [HttpGetClientCredentials](#), [HttpGetClientDirectory](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

HttpCreateServerCredentials Function

```
BOOL WINAPI HttpCreateServerCredentials(  
    DWORD dwProtocol,  
    DWORD dwOptions,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName,  
    LPVOID lpvReserved,  
    LPSECURITYCREDENTIALS* lppCredentials  
);
```

The **HttpCreateServerCredentials** function creates a SECURITYCREDENTIALS structure.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is

	supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A pointer to a string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A pointer to a string which specifies the certificate owner's password. A value of NULL specifies

that no password is required. This parameter is only used if a PKCS12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A pointer to a string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

lpvReserved

Pointer reserved for future use. Set it to NULL when using this function.

lppCredentials

Pointer to an [LPSECURITYCREDENTIALS](#) pointer. The memory for the credentials structure will be allocated by this function and must be released by calling the **HttpDeleteServerCredentials** function when it is no longer needed. The pointer value must be set to NULL before the function is called. It is important to note that this is a pointer to a pointer variable, not a pointer to the SECURITYCREDENTIALS structure itself.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The structure that is created by this function may be used as client credentials when establishing a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can

be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **HttpServerStart** function.

Example

```
// Create the server credentials that identifies the certificate
// that will be used for secure connections
LPSECURITYCREDENTIALS lpSecCred = NULL;

HttpCreateServerCredentials(SEcurity_PROTOCOL_DEFAULT,
                           0,
                           NULL,
                           NULL,
                           lpCertStore,
                           lpCertName,
                           NULL,
                           &lpSecCred);

// Start the server
hServer = HttpServerStart(lpLocalHost,
                          HTTP_PORT_SECURE,
                          HTTP_SERVER_SECURE,
                          &httpConfig,
                          lpEventHandler,
                          0,
                          lpSecCred);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpDeleteServerCredentials](#), [HttpServerStart](#), [SECURITYCREDENTIALS](#)

HttpCheckVirtualPath Function

```
BOOL WINAPI HttpCheckVirtualPath(  
    HSERVER hServer,  
    UINT nClientId,  
    LPCTSTR lpszVirtualPath,  
    DWORD dwFileAccess  
);
```

Determine if the client has permission to access the specified virtual path.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszVirtualPath

A pointer to a string which specifies the virtual path that should be checked. This path must be absolute and cannot be a NULL pointer or an empty string. The maximum length of the virtual path is 1024 characters.

dwFileAccess

An unsigned integer value which specifies the access permissions that should be checked. For a list of file access permissions, see [User and File Access Constants](#).

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid or the client ID does not specify a valid client, the function will return zero. If the function fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **HttpCheckVirtualPath** function is used to determine if the client has permission to access the virtual file or directory, based on the value of the *dwFileAccess* parameter. For example, if the *dwFileAccess* parameter has the value HTTP_ACCESS_WRITE, this function will check if the client has write permission for the file or directory. The function will return a non-zero value if the client does have the requested permission, or zero if it does not.

Applications that implement their own custom handlers for standard HTTP commands should use this function to ensure that the client has the appropriate permissions to access the requested resource. Failure to check the access permissions for the client can result in the client being able to access restricted documents and other resources. It is recommended that most applications use the default command handlers.

To obtain the path to the local file or directory that the virtual path is mapped to, use the **HttpGetClientLocalPath** function.

Example

```
TCHAR szPathName[1024];  
  
// Get the current request URL path  
INT cchPathName = HttpGetCommandResource(  
    hServer,
```

```
        nClientId,  
        szPathName,  
        1024);  
  
if (cchPathName == 0)  
{  
    HttpSendErrorResponse(hServer, nClientId, 500, NULL);  
    return;  
}  
  
// Check if the client has write access to that resource  
BOOL bAllowed = HttpCheckVirtualPath(  
    hServer,  
    nClientId,  
    szPathName,  
    HTTP_ACCESS_WRITE);  
  
if (!bAllowed)  
{  
    HttpSendErrorResponse(hServer, nClientId, 403, NULL);  
    return;  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAddVirtualPath](#), [HttpDeleteVirtualPath](#), [HttpGetClientLocalPath](#), [HttpGetClientVirtualPath](#)

HttpDeleteAllClientHeaders Function

```
BOOL WINAPI HttpDeleteAllClientHeaders(  
    HSERVER hServer,  
    UINT nClientId,  
    UINT nHeaderType  
);
```

Delete all of the request or response headers for the specified client session.

Parameters

hServer

The server handle.

nClientId

Delete all of the request or response headers for the specified client session.

nHeaderType

Specifies the type of headers to delete. It may be one of the following values:

Constant	Description
HTTP_HEADERS_REQUEST	Delete all of the request headers that were provided by the client. Request header values provide additional information to the server about the type of request being made.
HTTP_HEADERS_RESPONSE	Delete all of the response headers that were created by the server in response to a request made by the client. Response header values provide additional information to the client about the type of information that is being returned by the server.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **HttpDeleteAllClientHeaders** function is used to delete all of the request or response headers that were set as the result of the client issuing a request for a document or other resource. If this function is used to delete all of the response headers, the server will automatically generate a standard set of response headers when it returns the requested information to the client.

It is not necessary to call this function inside an HTTP_CLIENT_DISCONNECT event handler to delete the header values that were set during the client session. This is done automatically when the client disconnects from the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

See Also

[HttpDeleteClientHeader](#), [HttpSetClientHeader](#)

HttpDeleteClientHeader Function

```
BOOL WINAPI HttpDeleteClientHeader(  
    HSERVER hServer,  
    UINT nClientId,  
    UINT nHeaderType,  
    LPCTSTR lpszHeaderName  
);
```

Delete a request or response header for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

nHeaderType

Specifies the type of header to delete. It may be one of the following values:

Constant	Description
HTTP_HEADERS_REQUEST	Delete a request header that was provided by the client. Request header values provide additional information to the server about the type of request being made.
HTTP_HEADERS_RESPONSE	Delete a response header that was created by the server. Response header values provide additional information to the client about the type of information that is being returned by the server.

lpszHeaderName

A pointer to a string that specifies the name of the header that should be deleted. Header names are not case-sensitive and should not include the colon which acts as a delimiter that separates the header name from its value. This parameter cannot be a NULL pointer or an empty string.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero. If the function fails, the **HttpGetServerError** function will return more information about the last error that has occurred.

Remarks

The **HttpDeleteClientHeader** function will delete a request or response header for the specified client session. There are a number of required response headers that are always sent to a client and deleting the header using this function will cause the server to automatically generate a new default header value. You should not delete response header values unless you are certain of the impact that it would have on the normal operation of the client.

It is not necessary for you to delete a header value to change the value of an existing header. The **HttpSetClientHeader** function will replace an existing header value with a new value, or create a new header if the header name does not already exist.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpDeleteAllClientHeaders](#), [HttpGetClientHeader](#), [HttpSetClientHeader](#)

HttpDeleteServerCredentials Function

```
VOID WINAPI HttpDeleteServerCredentials(  
    LPSECURITYCREDENTIALS* LppCredentials  
);
```

The **HttpDeleteServerCredentials** function deletes an existing **SECURITYCREDENTIALS** structure.

Parameters

lppCredentials

Pointer to an LPSECURITYCREDENTIALS pointer. On exit from the function, the pointer value will be NULL.

Return Value

None.

Example

```
if (lpSecCred != NULL)  
    HttpDeleteServerCredentials(&lpSecCred);
```

Remarks

This function can be used to release the memory allocated for the credentials after the server has been started.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpCreateServerCredentials](#), [HttpServerStop](#)

HttpDeleteVirtualHost Function

```
BOOL WINAPI HttpDeleteVirtualHost(  
    HSERVER hServer,  
    UINT nHostId  
);
```

Delete a virtual host associated with the specified server.

Parameters

hServer

The server handle.

nHostId

An integer value which identifies the virtual host.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid or the virtual host ID does not specify a valid host, the function will return zero. If the function fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **HttpDeleteVirtualHost** function removes a virtual host that was created by a previous call to the **HttpAddVirtualHost** function. All virtual paths and users associated with the specified host are no longer valid. It is not necessary to call this function to delete any of the virtual hosts prior to stopping the server. Part of the normal shutdown process is releasing the resources allocated for each virtual host that was added to the server.

This function cannot be used to delete the virtual host with an ID of zero, which is the default virtual host that is allocated when the server is started.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[HttpAddVirtualHost](#), [HttpAddVirtualPath](#), [HttpAddVirtualUser](#) [HttpDeleteVirtualPath](#), [HttpDeleteVirtualUser](#)

HttpDeleteVirtualPath Function

```
BOOL WINAPI HttpDeleteVirtualPath(  
    HSERVER hServer,  
    UINT nHostId,  
    LPCTSTR lpszVirtualPath  
);
```

Remove a virtual path from the specified host.

Parameters

hServer

The server handle.

nHostId

An integer value which identifies the virtual host.

lpszVirtualPath

A pointer to a string which specifies the virtual path that will be removed. This path must be absolute and cannot be a NULL pointer or an empty string.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid or the virtual host ID does not specify a valid host, the function will return zero. If the function fails, the last error code will be updated to indicate the cause of the failure.

Remarks

This function removes a virtual path that was created by a previous call to the **HttpAddVirtualPath** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAddVirtualHost](#), [HttpAddVirtualPath](#), [HttpGetClientLocalPath](#), [HttpGetClientVirtualPath](#)

HttpDeleteVirtualUser Function

```
BOOL WINAPI HttpDeleteVirtualUser(  
    HSERVER hServer,  
    UINT nHostId,  
    LPCTSTR LpszUserName  
);
```

Remove a virtual user from the specified host.

Parameters

hServer

The server handle.

nHostId

An integer value which identifies the virtual host.

lpszUserName

A pointer to a string which specifies the user that will be removed. This parameter cannot be a NULL pointer or an empty string.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid or the virtual host ID does not specify a valid host, the function will return zero. If the function fails, the last error code will be updated to indicate the cause of the failure.

Remarks

This function removes a virtual user that was created by a previous call to the **HttpAddVirtualUser** function. This function will not match partial usernames and wildcard characters cannot be used to delete multiple users. Usernames are not case sensitive.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAddVirtualHost](#), [HttpAddVirtualUser](#)

HttpDisconnectClient Function

```
BOOL WINAPI HttpDisconnectClient(  
    HSERVER hServer,  
    UINT nClientId  
);
```

Close the control connection for the specified client and release the resources allocated for the session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **HttpDisconnectClient** function will close the control channel, disconnecting the client from the server and terminating the client session thread. Resources that we allocated for the client, such as memory and open handles, will be released back to the operating system. If the client was in the process of transferring a file, the transfer will be aborted.

This function sends an internal control message that notifies the server that this session should be terminated. When the session thread is signaled that it should terminate, it will abort any active data transfers and begin to release the resources allocated for that session. To ensure that the client session terminates gracefully, there may be a brief period of time where the session thread is still active after this function has returned.

After this function returns, the application should never use the same client ID with another function. Client IDs are unique to the session over the lifetime of the server, and are not reused.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[HttpServerRestart](#), [HttpServerStop](#)

HttpEnableClientAccess Function

```
BOOL WINAPI HttpEnableClientAccess(  
    HSERVER hServer,  
    UINT nClientId,  
    DWORD dwUserAccess,  
    BOOL bEnable  
);
```

Enable or disable access rights for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies an access right to enable or disable. For a list of user access rights that can be granted to the client, see [User and File Access Constants](#).

bEnable

An integer value which specifies if permission should be granted or revoked for the specified access right. If this value is non-zero, permission is granted to the client to perform the action specified by the *dwUserAccess* parameter. If this value is zero, that permission is revoked.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **HttpEnableClientAccess** function is used to enable or disable access to specific functionality by the client. The function can only change a single access right and cannot be used to enable or disable multiple access rights in a single function call. To change multiple user access rights for the client, use the **HttpSetClientAccess** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

See Also

[HttpAuthenticateClient](#), [HttpGetClientAccess](#), [HttpSetClientAccess](#)

HttpEnableCommand Function

```
BOOL WINAPI HttpEnableCommand(  
    HSERVER hServer,  
    LPCTSTR LpszCommand,  
    BOOL bEnable  
);
```

Enable or disable a specific server command

Parameters

hServer

The server handle.

lpszCommand

A pointer to a NULL terminated string that specifies the name of the command to be enabled or disabled. The command name is not case-sensitive, but the value must otherwise match the exact name. Partial matches are not recognized by this function. This parameter cannot be NULL.

bEnable

An integer value which specifies if the command should be enabled or disabled. If the value is non-zero, the command is enabled. If the value is zero, the command will be disabled.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid or the command is not recognized, the function will return zero. If the function fails, the **HttpGetServerError** function will return more information about the last error that has occurred.

Remarks

The **HttpEnableCommand** function is used to enable or disable access to a specific command on the server. This function is typically used to enable or disable certain commands for security purposes. For example, the PUT command can be disabled, preventing any client from attempting to upload files directly to the server. The **HttpIsCommandEnabled** function can be used to determine if a command is enabled or not.

The command name provided to this function must match the commands defined in RFC 2616 or related protocol standards. Refer to [Hypertext Transfer Protocol Commands](#) for a complete list of server commands.

Some commands cannot be disabled because they are required to perform essential server functions. For example, the GET and HEAD commands cannot be disabled. If you attempt to disable a required command, this function will return zero and the last error code will be set to ST_ERROR_COMMAND_REQUIRED. Because this function affects all clients connected to the server, it should not be used to limit access to certain commands for specific clients. Instead, use an event handler to filter the commands.

The OPTIONS and TRACE commands are disabled by default for all server instances and must be explicitly enabled using the **EnableCommand** method if you wish permit clients to use them. It is not recommended that you enable these commands if your server is going to be publicly accessible over the Internet. If the server started with the option HTTP_SERVER_READONLY, commands that can be used to create or modify files on the server will be disabled by default.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAuthenticateClient](#), [HttpIsCommandEnabled](#)

HttpEnumServerClients Function

```
INT WINAPI HttpEnumServerClients(  
    HSERVER hServer,  
    DWORD dwReserved  
    UINT * lpClients,  
    INT nMaxClients  
);
```

Return a list of active client sessions established with the specified server.

Parameters

hServer

Handle to the server socket.

dwReserved

An unsigned integer value that is reserved for future use. This parameter should always be zero.

lpClients

Pointer to an array of unsigned integers which will contain client IDs that uniquely identifies each client when the function returns. If this parameter is NULL, then the function will return the number of active client connections established with the server.

nMaxClients

Maximum number of client IDs to be returned in the *lpClients* array. If the *lpClients* parameter is NULL, this parameter should have a value of zero.

Return Value

If the function succeeds, the return value is the number of active client connections to the server. If the function fails, the return value is HTTP_ERROR. To get extended error information, call

HttpGetServerError.

Remarks

If the *nMaxClients* parameter is less than the number of active client connections, the function will fail and the last error code will be set to the error ST_ERROR_BUFFER_TOO_SMALL. To dynamically determine the number of active connections, call the function with the *lpClients* parameter with a value of NULL, and the *nMaxClients* parameter with a value of zero.

Example

```
INT nClients = HttpEnumServerClients(hServer, 0, NULL, 0);  
  
if (nClients > 0)  
{  
    UINT *lpClients = NULL;  
  
    // Allocate memory for an array of client IDs  
    lpClients = (UINT *)LocalAlloc(LPTR, nClients * sizeof(UINT));  
  
    if (lpClients == NULL)  
    {  
        // Virtual memory has been exhausted  
        return;  
    }  
  
    nClients = HttpEnumServerClients(hServer, 0, lpClients, nClients);
```

```
if (nClients == HTTP_ERROR)
{
    // Unable to obtain list of connected clients
    return;
}

for (INT nIndex = 0; nIndex < nClients; nIndex++)
{
    TCHAR szUserName[HTTP_MAXUSERNAME];

    if (HttpGetClientUserName(hServer, lpClients[nIndex], szUserName,
HTTP_MAXUSERNAME))
    {
        // Perform some action with the client user name
    }
}

// Free the memory allocated for the client IDs
LocalFree((HLOCAL)lpClients);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

See Also

[HttpServerStart](#)

HttpGetActiveClient Function

```
UINT WINAPI HttpGetActiveClient(  
    HSERVER hServer  
);
```

Return the client ID for the active client session associated with the current thread.

Parameters

hServer

The server handle.

Return Value

If the function succeeds, the return value is the unique ID associated with the client session for the current thread. If the server handle is invalid or there is no client session active on the current thread, the return value is zero.

Remarks

The **HttpGetActiveClient** function is used to obtain the client ID associated with the current thread. This means this function will only return a client ID if it is called within an event handler or a function called by an event handler. If this function is called by a function that is not executing within the context of an event handler it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[HttpEnumServerClients](#)

HttpGetClientAccess Function

```
BOOL WINAPI HttpGetClientAccess(  
    HSERVER hServer,  
    UINT nClientId,  
    LPDWORD LpdwUserAccess  
);
```

Return the access rights that have been granted to the client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpdwUserAccess

A pointer to an unsigned integer which specifies one or more access rights for the client session. For a list of user access permissions that can be granted to the client, see [User and File Access Constants](#). This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero. This function can only be used with authenticated clients. If the client session has not been authenticated, the return value will be zero.

Remarks

The **HttpGetClientAccess** function is used to obtain all of the access rights that are currently granted to the client session. The **HttpEnableClientAccess** function can be used to enable or disable specific permissions, and the **HttpSetClientAccess** function can change multiple access rights at once.

Example

```
DWORD dwUserAccess = 0;  
  
// Check if the client has execute permission  
if (HttpGetClientAccess(hServer, nClientId, &dwUserAccess))  
{  
    if (dwUserAccess & HTTP_ACCESS_EXECUTE)  
    {  
        _tprintf(_T("Client %u can execute programs and scripts\n"), nClientId);  
        return;  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

See Also

[HttpAuthenticateClient](#), [HttpEnableClientAccess](#), [HttpSetClientAccess](#)

HttpGetClientAddress Function

```
INT WINAPI HttpGetClientAddress(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

Return the IP address of the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszAddress

A pointer to a string buffer that will contain the client IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the server handle or the client ID is invalid, or the buffer is not large enough to store the complete address, the function will return a value of zero.

Remarks

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetServerAddress](#)

HttpGetClientCredentials Function

```
BOOL WINAPI HttpGetClientCredentials(  
    HSERVER hServer,  
    UINT nClientId,  
    LPHTTPCLIENTCREDENTIALS LpCredentials  
);
```

Return the user credentials for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpCredentials

A pointer to an **HTTPCLIENTCREDENTIALS** structure that will contain information about the user when the function returns. This parameter cannot be NULL.

Return Value

If the user credentials for the client session are available, the return value is non-zero. If the server handle and client ID do not specify a valid client session, or the client has not requested authentication, this function will return zero.

Remarks

The **HttpGetClientCredentials** function is used to obtain the username and password that was provided by the client when it requested authentication. Typically this function is used in an event handler to validate the credentials provided by the client. If the credentials are considered valid, the event handler would then call the **HttpAuthenticateClient** function to specify that the session has been authenticated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAuthenticateClient](#), [HttpRequireAuthentication](#), [HTTPCLIENTCREDENTIALS](#)

HttpGetClientDirectory Function

```
INT WINAPI HttpGetClientDirectory(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR lpszDirectory,  
    INT nMaxLength  
);
```

Returns the root document directory for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszDirectory

A pointer to a string buffer that will contain the root directory for the specified client session, terminated with a null character. This buffer should be at least MAX_PATH characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be larger than zero or the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

This function returns the full path to the root document directory for the specified client session. If no virtual hosts have been configured, then this path will be the same as the root directory assigned to the server when it was started. If the server has been configured with multiple virtual hosts, this function will return the path to the root directory associated with the hostname provided by the client.

To convert a full path to the virtual path for a specific client session, use the **HttpGetClientVirtualPath** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetClientVirtualPath](#)

HttpGetClientHeader Function

```
INT WINAPI HttpGetClientHeader(  
    HSERVER hServer,  
    UINT nClientId,  
    UINT nHeaderType,  
    LPCTSTR lpszHeaderName,  
    LPTSTR lpszHeaderValue,  
    INT nMaxLength,  
);
```

Return the value of a request or response header for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

nHeaderType

Specifies the type of header. It may be one of the following values:

Constant	Description
HTTP_HEADERS_REQUEST	Return the value of a request header that was provided by the client. Request header values provide additional information to the server about the type of request being made.
HTTP_HEADERS_RESPONSE	Return the value of a response header that was created by the server. Response header values provide additional information to the client about the type of information that is being returned by the server.

lpszHeaderName

A pointer to a string that specifies the name of the header field. Header names are not case-sensitive and should not include the colon which acts as a delimiter that separates the header name from its value. This parameter cannot be a NULL pointer or an empty string.

lpszHeaderValue

A pointer to a buffer that will contain the header value, terminated with a null character. To determine the length of the header value, this parameter can be NULL and the *nMaxLength* parameter should be specified with a value of zero.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the header value buffer, including the terminating null character. If the *lpszHeaderValue* parameter is NULL, this value must be zero.

Return Value

If the function succeeds, the return value is the length of the header value, not including the terminating null character. If the server handle and client ID do not specify a valid client session, or there is no header that matches the given name, the function will return zero. If the

lpzHeaderValue parameter is not NULL and the buffer is not large enough to store the complete header value, the function will return zero and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL. If the function fails, the **HttpGetServerError** function will return more information about the last error that has occurred.

Remarks

The **HttpGetClientHeader** function will return the value of a request or response header for the specified client session. If the *lpzHeaderName* value matches an existing header field, its value will be copied to the string buffer provided by the caller.

Refer to [Hypertext Transfer Protocol Headers](#) for a list of common request and response headers that are used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpDeleteClientHeader](#), [HttpGetAllClientHeaders](#), [HttpSetClientHeader](#)

HttpGetClientIdleTime Function

```
UINT WINAPI HttpGetClientIdleTime(  
    HSERVER hServer,  
    UINT nClientId,  
    UINT * lpnElapsed  
);
```

Return the idle timeout period for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpnElapsed

A pointer to an unsigned integer value that will contain the number of seconds the client session has been idle. This parameter may be NULL if this information is not required.

Return Value

If the function succeeds, the return value is client idle timeout period in seconds. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **HttpGetClientIdleTime** function will return the number of seconds that the client may remain idle before being automatically disconnected by the server. The idle time of a client session is based on the last time a command was issued to the server or when a data transfer completed. The server will never disconnect a client that is in the process of sending or receiving data, regardless of the idle timeout period.

The default idle timeout period for a client session is 60 seconds, however the server can be configured to use a different value. The minimum timeout period for a client is 10 seconds, the maximum is 300 seconds (5 minutes). An application can change the timeout period for a specific client session using the **HttpSetClientIdleTime** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

See Also

[HttpSetClientIdleTime](#)

HttpGetClientLocalPath Function

```
INT WINAPI HttpGetClientLocalPath(  
    HSERVER hServer,  
    UINT nClientId,  
    LPCTSTR lpszVirtualPath,  
    LPTSTR lpszLocalPath,  
    INT nMaxLength,  
);
```

Return the full local path for a virtual filename or directory on the server.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszVirtualPath

A pointer to a string that specifies a virtual path on the server. This parameter cannot be NULL.

lpszLocalPath

A pointer to a string buffer that will contain the full local path, terminated with a null-character. This buffer should be at least MAX_PATH characters to accommodate the complete path. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero. If the string buffer is not large enough to contain the complete path, this function will return zero and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL.

Remarks

The **HttpGetClientLocalPath** function takes a virtual path and returns the full path to the specified file or directory on the local system. The virtual path may be absolute or relative to the root directory for the client session.

To obtain the virtual path for a local file or directory, use the **HttpGetClientVirtualPath** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetClientVirtualPath](#)

HttpGetClientServer Function

```
HSERVER WINAPI HttpGetClientServer(  
    UINT nClientId  
);
```

The **HttpGetClientServer** function returns a handle to the server that created the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the function succeeds, the return value is the handle to the server that created the client session. If the function fails, the return value is `INVALID_SERVER`. To get extended error information, call the **HttpGetServerError** function.

Remarks

The **HttpGetClientServer** function returns the handle to the server that created the client session and is typically used within a notification message handler. If the server is in the process of shutting down, or the client session thread is terminating, this function will fail and return `INVALID_SERVER` indicating that the session ID is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cstools10.h`

Import Library: `cshtsv10.lib`

See Also

[HttpServerAsyncNotify](#)

HttpGetClientThreadId Function

```
DWORD WINAPI HttpGetClientThreadId(  
    HSERVER hServer,  
    UINT nClientId  
);
```

Returns the thread ID associated with the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the function succeeds, the return value is a thread ID. If the function fails, the return value is zero. To get extended error information, call the **HttpGetServerError** function.

Remarks

The **HttpGetClientThreadId** function returns a thread ID that can be used to identify the thread that is managing the client session. The thread ID can be used with other Windows API functions such as **OpenThread**. Exercise caution when using thread-related functions, interfering with the normal operation of the thread can have unexpected results. You should never use this function to obtain a thread handle and then call the **TerminateThread** function to terminate a client session. This will prevent the thread from releasing the resources that were allocated for the session and can leave the server in an unstable state. To terminate a client session, use the **HttpDisconnectClient** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshtsv10.lib`

See Also

[HttpEnumServerClients](#), [HttpGetActiveClient](#)

HttpGetClientUserName Function

```
INT WINAPI HttpGetClientUserName(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR lpszUserName,  
    INT nMaxLength  
);
```

Return the user name associated with the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszUserName

A pointer to a string buffer that will contain the user name associated with the client session. This buffer must be large enough to store the complete user name, including the terminating null character. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This parameter must have a value larger than zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **HttpGetClientUserName** function returns the user name associated with an authenticated client session. If the client has not authenticated itself, this function will return zero and the *lpszUserName* parameter will be set to an empty string.

The **HttpIsClientAuthenticated** function can be used to determine if the client has provided credentials as part of the request made to the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAuthenticateClient](#), [HttpGetClientAccess](#), [HttpIsClientAuthenticated](#)

HttpGetClientVariable Function

```
INT WINAPI HttpGetClientVariable(  
    HSERVER hServer,  
    UINT nClientId,  
    LPCTSTR lpszName,  
    LPTSTR lpszValue,  
    INT nMaxLength,  
);
```

Return the value of a CGI environment variable for the specified client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszName

A pointer to a string that specifies the name of the environment variable. Variable names are not case-sensitive and should not include the equal sign which acts as a delimiter that separates the variable name from its value. This parameter cannot be a NULL pointer or an empty string.

lpszValue

A pointer to a buffer that will contain the value of the environment variable, terminated with a null character. To determine the length of the header value, this parameter can be NULL and the *nMaxLength* parameter should be specified with a value of zero.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the value buffer, including the terminating null character. If the *lpszValue* parameter is NULL, this value must be zero.

Return Value

If the function succeeds, the return value is the length of the environment variable value, not including the terminating null character. If the server handle and client ID do not specify a valid client session, or there is no environment variable that matches the given name, the function will return zero. If the *lpszValue* parameter is not NULL and the buffer is not large enough to store the complete header value, the function will return zero and the last error code will be set to `ST_ERROR_BUFFER_TOO_SMALL`. If the function fails, the **HttpGetServerError** function will return more information about the last error that has occurred.

Remarks

The **HttpGetClientVariable** function will return the value of an environment variable that has been defined for the client. Each client session inherits a copy of the process environment block, which is then modified to define various environment variables that are used with CGI programs and scripts. The **HttpSetClientVariable** function can be used to change existing environment variables or create new variables.

The standard CGI environment variables that are defined by the server are not created until the client request has been processed. This means that environment variables such as `REMOTE_ADDR` and `SERVER_NAME` will not be defined inside an `HTTP_CLIENT_CONNECT` event handler.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpSetClientVariable](#)

HttpGetClientVirtualHost Function

```
INT WINAPI HttpGetClientVirtualHost(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR lpszHostName,  
    INT nMaxLength  
);
```

Return the name of the virtual host the client used to establish the connection.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszHostName

A pointer to a string buffer that will contain the virtual host name. The string buffer will be null terminated and must be large enough to store the complete hostname. If this parameter is NULL, the function will only return the length of the current command in characters, not including the terminating null character.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. If the maximum length specified is smaller than the actual length of the parameter, this function will fail. If the *lpszHostName* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the function fails, the return value will be zero and the **HttpGetServerError** function can be used to retrieve the last error code.

Remarks

The **HttpGetClientVirtualHost** function is used to obtain the hostname that the client used to establish a connection with the server. This function is typically used within an event handler to determine the hostname associated with the request made by the client. It should not be called inside a HTTP_CLIENT_CONNECT event handler because the virtual host has not been selected at that point. If the virtual hostname is not available at the time this function is called, the function will return zero and the last error code will be set to ST_ERROR_VIRTUAL_HOST_NOT_FOUND.

The **HttpGetClientVirtualHostId** function can be used to obtain the virtual host ID associated with the hostname.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetClientVirtualHostId](#), [HttpGetCommandUrl](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

HttpGetClientVirtualHostId Function

```
UINT WINAPI HttpGetClientVirtualHostId(  
    HSERVER hServer,  
    UINT nClientId  
);
```

Return the virtual host ID associated with the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

An unsigned integer value which specifies the virtual host ID. If the function fails, the return value will be `INVALID_VIRTUAL_HOST` and the **HttpGetServerError** function can be used to retrieve the last error code.

Remarks

The **HttpGetClientVirtualHostId** function is used to obtain the virtual host ID associated with the hostname the client used to establish a connection with the server. This function should not be called inside a `HTTP_CLIENT_CONNECT` event handler because the virtual host has not been selected at that point. If the virtual host ID is not available at the time this function is called, the function will return `INVALID_VIRTUAL_HOST` and the last error code will be set to `ST_ERROR_VIRTUAL_HOST_NOT_FOUND`.

The **HttpGetClientVirtualHost** function can be used to obtain the hostname that the client used to establish the connection.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshtsv10.lib`

See Also

[HttpGetClientVirtualHost](#), [HttpGetCommandUrl](#), [HttpGetVirtualHostId](#), [HttpGetVirtualHostName](#)

HttpGetClientVirtualPath Function

```
INT WINAPI HttpGetClientVirtualPath(  
    HSERVER hServer,  
    UINT nClientId,  
    LPCTSTR lpszLocalPath,  
    LPTSTR lpszVirtualPath,  
    INT nMaxLength,  
);
```

Return the virtual path for a local file on the server.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszLocalPath

A pointer to a string that specifies an absolute path on the local system. This parameter cannot be NULL.

lpszVirtualPath

A pointer to a string buffer that will contain the virtual path, terminated with a null-character. This buffer should be at least MAX_PATH characters to accommodate the complete path. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero. If the string buffer is not large enough to contain the complete path, this function will return zero and the last error code will be set to ST_ERROR_BUFFER_TOO_SMALL.

Remarks

A virtual path for the client is relative to the root directory for the specified client session. These virtual paths are what the client will see as an absolute path on the server. For example, if the server was configured to use "C:\ProgramData\MyServer" as the root directory, and the *lpszLocalPath* parameter was specified as "C:\ProgramData\MyServer\Documents\Research", this function would return the virtual path to that directory as "/Documents/Research".

If the *lpszLocalPath* parameter specifies a file or directory outside of the server root directory, this function will return zero and the last error code will be set to ST_ERROR_INVALID_FILE_NAME. This function can only be used with authenticated clients. If the *nClientId* parameter specifies a client session that has not been authenticated, this function will return zero and the last error code will be ST_ERROR_AUTHENTICATION_REQUIRED.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetCommandFile](#), [HttpGetClientLocalPath](#)

HttpGetCommandFile Function

```
INT WINAPI HttpGetCommandFile(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR LpszFileName,  
    INT nMaxLength  
);
```

Get the full path to a file name or directory specified by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A pointer to a string buffer that will contain the full path to a file name or directory specified by the client when it issued a command. The string buffer will be null terminated and must be large enough to store the complete file path. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. It is recommended that the buffer be at least MAX_PATH characters in size. If the maximum length specified is smaller than the actual length of the full path, this function will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the function fails, the return value will be zero and the **HttpGetServerError** function can be used to retrieve the last error code.

Remarks

The **HttpGetCommandFile** function is used to obtain the full path to a local file name or directory specified by the client as an argument to a standard HTTP command. For example, if the client sends the GET command to the server, this function will return the complete path to the local file that the client wants to retrieve. This function will only work with those standard commands that perform some action on a file or directory.

This function should always be used to obtain the file name for a command that performs a file or directory operation. It normalizes the path provided by the client and ensures that it specifies a file or directory name in the correct location. The **HttpGetCommandUrl** function can be used to obtain the URL that was provided by the client.

To map a virtual path to a file or directory on the local system, use the **HttpAddVirtualPath** function. To redirect a client to use a different URL to access the resource, use the **HttpRedirectRequest** function.

The **HttpSetCommandFile** function can be used to change the name of the local file or directory that is the target of the command, however using this function to redirect access to a resource can have unintended side-effects, particularly in the case where the URL provided by the client actually resolves to an executable CGI program that handles the request.

If the client has provided a URL that resolves to a CGI program that handles the request, this

function behaves differently than if the URL is resolved to a local file or directory. If the client uses the GET or POST command that results in a program being executed to handle the request, this function will return the path to the server root directory along with any additional path information provided in the URL. In other words, the file name returned by this function will be the same as the PATH_TRANSLATED value passed to the CGI program.

This function should only be called after the client request has been received by the server, typically inside a HTTP_CLIENT_COMMAND or HTTP_CLIENT_EXECUTE event handler. It should not be called inside a HTTP_CLIENT_CONNECT event handler because the server has not processed the client request at that point.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAddVirtualPath](#), [HttpGetCommandUrl](#), [HttpRedirectRequest](#), [HttpSetCommandFile](#)

HttpGetCommandLine Function

```
INT WINAPI HttpGetCommandLine(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR lpszCmdLine,  
    INT nMaxLength  
);
```

Return the complete command line issued by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCmdLine

A pointer to a string buffer that will contain the command, including all arguments. The string buffer will be null terminated and must be large enough to store the complete command line. If this parameter is NULL, the function will return the length of the command line.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. The internal limit on the maximum length of a command is 1024 characters. If the maximum length specified is smaller than the actual length of the complete command, this function will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the function fails, the return value will be zero and the

HttpGetServerError function can be used to retrieve the last error code. If the last error code has a value of zero then no command has been issued by the client.

Remarks

The **HttpGetCommandLine** function is used to obtain the command that was issued by the client, and is commonly used inside HTTP_CLIENT_COMMAND and HTTP_CLIENT_RESULT event handlers to pre-process and post-process client commands, respectively. When the function returns, the string buffer provided by the caller will contain the complete command, including the resource path and requested HTTP version. Any extraneous whitespace will be removed, however all encoding will be preserved.

To obtain the complete URL associated with the request issued by the client, use the **HttpGetCommandUrl** function. If the command sent by the client is used to perform an action on a file or directory, the **HttpGetCommandFile** function should be called to obtain the full path to the local file rather than using the resource path.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetCommandFile](#), [HttpGetCommandUrl](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

HttpGetCommandName Function

```
INT WINAPI HttpGetCommandName
    HSERVER hServer,
    UINT nClientId,
    LPTSTR LpszCommand,
    INT nMaxLength
);
```

Return the name of the last command issued by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCommand

A pointer to a string buffer that will contain the command name. The string buffer will be null terminated and must be large enough to store the complete command name. If this parameter is NULL, the function will only return the length of the current command in characters, not including the terminating null character.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. If the maximum length specified is smaller than the actual length of the parameter, this function will fail. If the *lpszCommand* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the function fails, the return value will be zero and the **HttpGetServerError** function can be used to retrieve the last error code. If the last error code is returned as a value of zero, this means that no command has been issued by the client.

Remarks

The **HttpGetCommandName** function is used to obtain the name of the last command that was issued by the client. The command name returned by this function will always be capitalized, regardless of how it was sent by the client. This function is typically used inside HTTP_CLIENT_COMMAND and HTTP_CLIENT_RESULT event handlers to pre-process and post-process client commands, respectively. It should not be called inside a HTTP_CLIENT_CONNECT event handler because the server has not processed the client request at that point.

The **HttpGetCommandUrl** function can be used to return the resource that was requested by the client.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetCommandFile](#), [HttpGetCommandUrl](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

HttpGetCommandQuery Function

```
INT WINAPI HttpGetCommandQuery(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR lpszParameters,  
    INT nMaxLength  
);
```

Return the query parameters included with the command.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszParameters

A pointer to a string buffer that will contain the query parameters when the function returns. The string buffer will be null terminated up to the maximum number of characters specified by the caller. If this parameter is NULL the function will return the length of the query string.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. If the *lpszParameters* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the function fails, the return value will be zero and the

HttpGetServerError function can be used to retrieve the last error code. If the client did not provide any query parameters, this function will return zero and the last error code will be zero.

Remarks

The **HttpGetCommandQuery** function is used to obtain a copy of the query parameters that were included in the request URL. If there were no query parameters, the string buffer will be empty and the return value will be zero. If the request did include query parameters, they will be returned to the caller in their original, encoded form.

This function should not be called within a HTTP_CLIENT_CONNECT event handler because the client request has not been processed at that point.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetCommandName](#), [HttpGetCommandResource](#), [HttpReceiveRequest](#)

HttpGetCommandResource Function

```
INT WINAPI HttpGetCommandResource(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR lpszResource,  
    INT nMaxLength  
);
```

Return the URL path for the resource requested by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszResource

A pointer to a string buffer that will contain the URL path provided by the client for the current command. The string buffer will be null terminated and must be large enough to store the complete path. If this parameter is NULL, the function will only return the length of the path.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. If the *lpszResource* parameter is NULL, this value should be zero. If this value is less than the length of the URL, the function will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the function fails, the return value will be zero and the **HttpGetServerError** function can be used to retrieve the last error code.

Remarks

The **HttpGetCommandResource** function returns the URL path that the client provided to access the requested resource. The path will not include the URI scheme, user credentials or query parameters. The **HttpGetCommandFile** function can be used to determine the name of the local file on the server that will be accessed using this URL.

If you require the complete URL, not just the path to the resource, use the **HttpGetCommandUrl** function.

This function should only be called after the client request has been received by the server, typically inside a HTTP_CLIENT_COMMAND event handler. It should not be called inside a HTTP_CLIENT_CONNECT event handler because the server has not processed the client request at that point.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetCommandFile](#), [HttpGetCommandQuery](#), [HttpGetCommandUrl](#), [HttpRedirectRequest](#)

HttpGetCommandResult Function

```
INT WINAPI HttpGetCommandResult(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR lpszResult,  
    INT nMaxLength  
);
```

Return the result code and description for the last command issued by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszResult

A pointer to a string buffer that will contain the description of the result code. The string buffer will be null terminated up to the maximum number of characters specified by the caller. This parameter can be NULL if this information is not required.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. If the *lpszResult* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the result code for the last command issued by the client. A return value of zero indicates that the command has not completed and there is no result code available.

Remarks

The **HttpGetCommandResult** function is used to determine the result of the last command that was issued by the client and is typically called in the HTTP_CLIENT_RESULT event handler. This function should only be called after a command has been processed or the **HttpSendResponse** function has been called.

The result code is a three-digit integer value that indicates the success or failure of a command. Whenever a client sends a command to the server, the server must respond with this numeric code and a brief description of the the result. Result codes are generally broken down into the following categories:

Result Code	Description
100-199	Result codes in this range are informational and only used with version 1.1 of the protocol. If the server returns a result code in this range, it means that it has received the request and that the client should proceed.
200-299	Result codes in this range indicate that the server has successfully completed the requested action. In most cases this means that the server has returned the requested data to the client, however if a 204 result code is sent, this indicates that the request has been processed but there is no data available.
300-399	Result codes in this range indicate that the requested resource has been moved to a new location. The most common result codes are 301 and 302. A value of

	301 indicates that the location of the resource has changed permanently and all future requests should be sent to the new URL. A value of 302 indicates that the resource location has changed temporarily. The new location of the resource is sent to the client by setting the Location response header field.
400-499	Result codes in this range indicate an error with the request that was made by the client. These types of errors include invalid commands, requests that can only be issued by authenticated clients, or resources that cannot be accessed on the server. The most common result code in this range is the 404 code which indicates that the requested document could not be found.
500-599	Result codes in this range indicate a server error has occurred while processing a valid request. These types of errors are returned when a command has not been implemented, the execution of a CGI program or script has failed unexpectedly, or an internal server error has occurred.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpReceiveRequest](#), [HttpSendResponse](#)

HttpGetCommandUrl Function

```
INT WINAPI HttpGetCommandUrl(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR lpszCmdUrl,  
    INT nMaxLength  
);
```

Return the complete URL of the resource requested by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszCmdUrl

A pointer to a string buffer that will contain the URL provided by the client for the current command. The string buffer will be null terminated and must be large enough to store the complete URL. If this parameter is NULL, the function will only return the length of the URL.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. If the *lpszCmdUrl* parameter is NULL, this value should be zero. If this value is less than the length of the URL, the function will fail.

Return Value

An integer value which specifies the number of characters copied into the buffer, not including the terminating null character. If the function fails, the return value will be zero and the **HttpGetServerError** function can be used to retrieve the last error code.

Remarks

The **HttpGetCommandUrl** function returns the complete URL that the client provided to access the requested resource. The URL will include any query parameters that were specified by the client, but it will not include any user credentials. The **HttpGetCommandFile** function can be used to determine the name of the local file on the server that will be accessed using this URL.

If you only require the URL path, without the URI scheme or the query parameters, use the **HttpGetCommandResource** function.

This function should only be called after the client request has been received by the server, typically inside a HTTP_CLIENT_COMMAND event handler. It should not be called inside a HTTP_CLIENT_CONNECT event handler because the server has not processed the client request at that point.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetCommandFile](#), [HttpGetCommandResource](#), [HttpRequestRedirectRequest](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

HttpGetProgramExitCode Function

```
BOOL WINAPI HttpGetProgramExitCode(  
    HSERVER hServer,  
    UINT nClientId,  
    LPDWORD lpdwExitCode  
);
```

Return the exit code of the last program executed by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpdwExitCode

A pointer to an unsigned integer that will contain the program exit code when the function returns. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **HttpGetProgramExitCode** function returns the exit code of a registered CGI program or script that was executed. By convention, most programs return an exit code in the range of 0-255, with an exit code of zero indicating success. The exit code is commonly used by programs to communicate status information back to the server application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

See Also

[HttpGetProgramOutput](#), [HttpRegisterProgram](#)

HttpGetProgramName Function

```
INT WINAPI HttpGetProgramName(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR lpszProgramName,  
    INT nMaxLength  
);
```

Return the name of the CGI program executed by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszProgramName

A pointer to a string buffer that will contain the name of the CGI program executed by the client. This parameter cannot be NULL and should be at least MAX_PATH characters in size.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer, including the terminating null character. This parameter must have a value greater than zero.

Return Value

If the function succeeds, the return value is the length of the executable file name. If the server handle and client ID do not specify a valid client session, the function will return zero. If the client has not executed a CGI program this function will return zero and the last error code will be set to zero.

Remarks

The **HttpGetProgramName** function returns the local file name of the CGI program that was executed in response to a request from the client. This is the full path to the executable that was registered with the server using the **HttpRegisterProgram** function. If the client specified a regular document or directory, this function will return a value of zero, indicating that no CGI program has been executed to handle the request.

To obtain the resource URL that was provided by the client, use the **HttpGetCommandUrl** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetProgramExitCode](#), [HttpGetProgramOutput](#), [HttpRegisterProgram](#)

HttpGetProgramOutput Function

```
DWORD WINAPI HttpGetProgramOutput(  
    HSERVER hServer,  
    UINT nClientId,  
    LPBYTE lpBuffer,  
    DWORD dwBufferSize  
);
```

Return a copy of the standard output from a CGI program executed by the client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpBuffer

A pointer to a buffer that will contain the output from the last program executed by the client. If this parameter is NULL, the function will return the number of bytes of data that was output by the program. Note that this output is not null terminated.

dwBufferSize

The maximum number of bytes that can be copied into the buffer. If the *lpBuffer* parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the number of bytes copied into the specified buffer. If the server handle and client ID do not specify a valid client session, the function will return zero. If the client has not executed any programs, the return value will be zero.

Remarks

The **HttpGetProgramOutput** function is used to obtain a copy of the output generated by a CGI program. To determine the number of bytes of output available to read, call this function with the *lpBuffer* parameter as NULL and the *dwBufferSize* parameter with a value of zero. The return value will be the number of bytes of data that was output by the program. It should be noted that for Unicode builds, the buffer is a byte array, not an array of characters, and will not be null terminated.

This function returns the raw output from the program which may contain a response header block, escape sequences, control characters and embedded nulls. When the application processes the output returned by this function, it should never coerce the buffer pointer to an LPTSTR value because there is no guarantee that the data will be null-terminated. To obtain the output from the program as a string, use the **HttpGetProgramText** function.

This function should only be used within an HTTP_CLIENT_EXECUTE event handler, which occurs after the program has terminated.

Example

```
LPBYTE lpBuffer = NULL; // A pointer to the output buffer  
DWORD cbBuffer = 0;     // Number of bytes in the output buffer  
  
// Determine the number of bytes in the output buffer
```

```
cbBuffer = HttpGetProgramOutput(hServer, nClientId, NULL, 0);

if (cbBuffer > 0)
{
    // Allocate memory for the buffer
    lpBuffer = (LPBYTE)LocalAlloc(LPTR, cbBuffer + 1);

    // Copy the program output to the buffer
    if (lpBuffer != NULL)
        cbBuffer = HttpGetProgramOutput(hServer, nClientId, lpBuffer, cbBuffer +
1);
}

// Free the memory allocated for the buffer when finished
if (lpBuffer != NULL)
{
    LocalFree((HLOCAL)lpBuffer);
    lpBuffer = NULL;
    cbBuffer = 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

See Also

[HttpGetProgramExitCode](#), [HttpGetProgramText](#), [HttpRegisterProgram](#)

HttpGetProgramText Function

```
INT WINAPI HttpGetProgramText(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);
```

Return a copy of the standard output from a CGI program in a string buffer.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszBuffer

A pointer to a buffer that will contain the output from the last program executed by the client as a string. If this parameter is NULL, the function will return the number of bytes of characters that was output by the program, not including a terminating null character.

nMaxLength

The maximum number of bytes that can be copied into the buffer. If the *lpszBuffer* parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the specified string buffer, not including the terminating null character. If the server handle and client ID do not specify a valid client session, the function will return zero. If the client has not executed any programs, the return value will be zero.

Remarks

The **HttpGetProgramText** function is used to obtain a copy of the output generated by a CGI program. To determine the number of characters of output available to read, call this function with the *lpszBuffer* parameter as NULL and the *nMaxLength* parameter with a value of zero. The return value will be the number of characters that were output by the program. If the application dynamically allocates the string buffer, make sure that it allocates an extra character for the terminating null character.

This function will only return textual output from the program and any non-printable control characters and the escape character will be replaced with a space. To obtain the unfiltered output from the program, use the **HttpGetProgramOutput** function. If the program outputs a response header block, this will be included in the string buffer.

This function should only be used within an HTTP_CLIENT_EXECUTE event handler, which occurs after the program has terminated.

Example

```
LPTSTR lpszBuffer = NULL; // A pointer to the output buffer  
INT cchBuffer = 0; // Number of bytes in the output buffer  
  
// Determine the number of bytes in the output buffer
```

```
cchBuffer = HttpGetProgramText(hServer, nClientId, NULL, 0);

if (cchBuffer > 0)
{
    // Allocate memory for the string buffer
    lpszBuffer = (LPTSTR)LocalAlloc(LPTR, (cchBuffer + 1) * sizeof(TCHAR));

    // Copy the program output to the buffer
    if (lpszBuffer != NULL)
        cchBuffer = HttpGetProgramText(hServer, nClientId, lpszBuffer, cchBuffer
+ 1);
}

// Free the memory allocated for the buffer when finished
if (lpszBuffer != NULL)
{
    LocalFree((HLOCAL)lpszBuffer);
    lpszBuffer = NULL;
    cchBuffer = 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetProgramExitCode](#), [HttpGetProgramOutput](#), [HttpRegisterProgram](#)

HttpGetServerAddress Function

```
INT WINAPI HttpGetServerAddress(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

Return the IP address of the server.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszAddress

A pointer to a string buffer that will contain the server IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the server handle or the client ID is invalid, or the buffer is not large enough to store the complete address, the function will return a value of zero.

Remarks

This function will return the IP address assigned to the specified server as a printable string. If the *nClientId* parameter has a value of zero, this function will return the IP address assigned to the local system. If the HTTP_SERVER_NATROUTER option was specified when the server was started, this function will return the external IP address assigned to the system. If the *nClientId* parameter specifies a valid client session, this function will return the IP address that the client used to establish the connection with the server. To determine the IP address assigned to the client, use the **HttpGetClientAddress** function.

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetClientAddress](#), [HttpGetServerName](#)

HttpGetServerDirectory Function

```
INT WINAPI HttpGetServerDirectory(  
    HSERVER hServer,  
    LPTSTR lpszDirectory,  
    INT nMaxLength  
);
```

Return the full path to the root directory assigned to the specified server.

Parameters

hServer

The server handle.

lpszDirectory

A pointer to a string buffer that will contain the server root directory, terminated with a null character. It is recommended that this buffer be at least MAX_PATH characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle is invalid, or the buffer is not large enough to store the complete path, the function will return a value of zero.

Remarks

The **HttpGetServerDirectory** function will return the full path to the root directory assigned to the server instance. The root directory may be specified as part of the server configuration, or if no directory is specified by the application, a temporary root directory will be created and this function can be used to obtain the full path to the directory. When the application specifies a root directory, it may use environment variables such as %AppData% in the path. This function will return the fully resolved path name, with all environment variables expanded.

There is no corresponding function to change the server root directory after the server has started. To change the root directory, you must stop the server using the **HttpServerStop** function and then start another instance of the server with a configuration that specifies the new directory.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetServerAddress](#), [HttpGetServerIdentity](#), [HttpGetServerName](#)

HttpGetServerError Function

```
DWORD WINAPI HttpGetServerError(  
    HSERVER hServer,  
    LPTSTR lpszError,  
    INT nMaxLength  
);
```

Return the last server error code and a description of the error.

Parameters

hServer

The server handle.

lpszError

A pointer to a string buffer that will contain a description of the error. If the error description is not needed, this parameter may be NULL.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the error string buffer, including the terminating null character. If the *lpszError* parameter is NULL, this value should be zero.

Return Value

An unsigned integer value that specifies the last error that occurred. A value of zero indicates that there was no error.

Remarks

Error codes are unsigned 32-bit values which are private to each server. You should call the **HttpGetServerError** function immediately when a function's return value indicates that an error has occurred. That is because some functions clear the last error code when they succeed.

If the *hServer* parameter is specified with a value of INVALID_SERVER, this function will return the last error that occurred for the current thread. This value should only be used when the function does not have access to a valid server handle, such as when the **HttpServerStart** function fails.

It is important to note that the error codes returned by this method are different than the command result codes that are defined in RFC 2616, the standard protocol specification for HTTP. This function is used to determine reason that an API function has failed, and should not be used to determine if a command issued by the client was successful. The **HttpSendResponse** function is used to send responses to the client, and the **HttpGetCommandResult** function can be used to determine the result of the last command sent by the client.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SERVER or HTTP_ERROR. Those functions which clear the last error code when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetCommandResult](#), [HttpSendResponse](#), [HttpSetServerError](#)

HttpGetServerIdentity Function

```
INT WINAPI HttpGetServerIdentity(  
    HSERVER hServer,  
    LPTSTR lpszIdentity,  
    INT nMaxLength  
);
```

Return the identity of the specified server.

Parameters

hServer

The server handle.

lpszIdentity

A pointer to a string buffer that will contain the identity of the server when the function returns, terminated with a null character. This parameter cannot be NULL. It is recommended that this buffer be at least 32 characters in length.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If the server handle is invalid, or the buffer is not large enough to store the complete path, the function will return a value of zero.

Remarks

The **HttpGetServerIdentity** function returns the identity string that was specified as part of the server configuration. It is used for informational purposes only and does not affect the operation of the server. Typically the string specifies the name of the application and a version number. The **HttpSetServerIdentity** function can be used to change the identity string associated with the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpSetServerIdentity](#)

HttpGetServerLogFile Function

```
BOOL WINAPI HttpGetServerLogFile(  
    HSERVER hServer,  
    UINT * lpnLogFormat,  
    UINT * lpnLogLevel,  
    LPTSTR lpszFileName,  
    INT nMaxLength  
);
```

Return the current log file format and the full path to the file.

Parameters

hServer

The server handle.

lpnLogFormat

A pointer to an integer value that will contain the log file format being used when the function returns. If this information is not needed, this parameter may be NULL. The following formats are supported:

Constant	Description
HTTP_LOGFILE_NONE (0)	This value specifies that the server should not create or update a log file.
HTTP_LOGFILE_COMMON (1)	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about file transfers.
HTTP_LOGFILE_COMBINED (2)	This value specifies that the server should use the combined log file format. This format is similar to the common format, however it includes the client referrer and user agent. This is the format that most Apache web servers use by default.
HTTP_LOGFILE_EXTENDED (3)	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session.

lpnLogLevel

A pointer to an integer value that will contain the level of detail the server uses when generating the log file. The minimum value is 1 and the maximum value is 10. If this information is not needed, this parameter may be NULL.

lpszFileName

A pointer to a string buffer that will contain the full path to the log file. This parameter may be NULL if this information is not required.

nMaxLength

An integer that specifies the maximum number of characters that can be copied into the file name string, including the terminating null character. If the *lpszFileName* parameter is NULL, this value should be zero.

Return Value

An integer value which specifies the current log file format. Refer to the **HTTPSERVERCONFIG** structure definition for a list of supported log file formats. If the server handle is invalid or logging has not been enabled, this function will return a value of zero.

Remarks

If the server is configured with logging enabled, but a log file name is not explicitly provided, then the server will automatically generate one. This function can be used to get the full path to the current log file along with the format that is being used to record client session data. Normally the log file is held open by the server thread while it is active, however you can call the **HttpRenameServerLogFile** function to explicitly rename or delete the log file.

To change the name of the log file, the log file format or level of detail, use the **HttpSetServerLogFile** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpRenameServerLogFile](#), [HttpSetServerLogFile](#)

HttpGetServerMemoryUsage Function

```
BOOL WINAPI HttpGetServerMemoryUsage(  
    HSERVER hServer,  
    ULARGE_INTEGER * LpMemUsage  
);
```

Return the amount of memory allocated for the server and all client sessions.

Parameters

hServer

The server handle.

lpMemUsage

A pointer to a ULARGE_INTEGER variable which will specify how much memory has been allocated by the server. This parameter will be initialized to a value of zero by the function and updated with the total number of bytes allocated by the server and all active client sessions when it returns.

Return Value

If the function succeeds, the return value is non-zero and the memory usage value will be updated. If the server handle is invalid, or the server cannot be locked, the return value is zero. Call the **HttpGetServerError** function to determine the cause of the failure.

Remarks

This function returns the amount of memory allocated by the server and all active client sessions. It enumerates all of memory allocations made by the server process and client session threads and returns the total number of bytes allocated for the server process. This value reflects the amount of memory explicitly allocated by this library and does not reflect the total working set size of the process, or memory allocated by any other libraries. To determine the working set size for the process, refer to the Win32 **GetProcessWorkingSetSize** and **GetProcessMemoryInfo** functions.

This function forces the server into a locked state, and all client sessions will block until the function returns. Because this function enumerates all heaps allocated for the server process, it can be an expensive operation, particularly when there are a large number of active clients connected to the server. Frequent use of this function can significantly degrade the performance of the server. It is primarily intended for use as a debugging tool to determine if memory usage is the result of an increase in active client sessions. If the value returned by the function remains reasonably constant, but the amount of memory allocated for the process continues to grow, it could indicate a memory leak in some other area of the application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetServerStackSize](#), [HttpSetServerStackSize](#)

HttpGetServerName Function

```
INT WINAPI HttpGetServerName(  
    HSERVER hServer,  
    UINT nClientId,  
    LPTSTR lpszHostName,  
    INT nMaxLength  
);
```

Return the host name assigned to the specified server.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszHostName

A pointer to a string buffer that will contain the server host name, terminated with a null character. It is recommended that this buffer be at least 64 characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the server handle or the client ID is invalid, or the buffer is not large enough to store the complete hostname, the function will return a value of zero.

Remarks

This function will return the host name assigned to the specified server. If the *nClientId* parameter has a value of zero, the function will return the default host name that was specified as part of the server configuration. If no host name was explicitly assigned to the server, then it will return the local system name. If the *nClientId* parameter specifies a client session, then it this function will return the host name that the client used to establish the connection.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetServerAddress](#)

HttpGetServerOptions Function

```
BOOL WINAPI HttpGetServerOptions(  
    HSERVER hServer,  
    LPDWORD LpdwOptions  
);
```

Return the configuration options for the specified server.

Parameters

hServer

The server handle.

LpdwOptions

A pointer to an unsigned integer that will contain the options that were specified when the server was started. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid, the function will return a value of zero.

Remarks

This function will return the options that were specified when the server was started. For a list of server options see [Server Option Constants](#).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshtsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpServerStart](#)

HttpGetServerPriority Function

```
INT WINAPI HttpGetServerPriority(  
    HSERVER hServer  
);
```

Return the current priority assigned to the specified server.

Parameters

hServer

The server handle.

Return Value

If the function succeeds, the return value is the priority for the specified server. If the function fails, the return value is HTTP_PRIORITY_INVALID. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpGetServerPriority** function can be used to determine the current priority assigned to the server. It will return one of the following values:

Constant	Description
HTTP_PRIORITY_BACKGROUND (0)	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
HTTP_PRIORITY_LOW (1)	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
HTTP_PRIORITY_NORMAL (2)	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
HTTP_PRIORITY_HIGH (3)	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
HTTP_PRIORITY_CRITICAL (4)	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cstools10.h

Import Library: cshtsv10.lib

See Also

[HttpServerStart](#), [HttpSetServerPriority](#)

HttpGetServerStackSize Function

```
DWORD WINAPI HttpGetServerStackSize(  
    SOCKET hServer  
);
```

Return the initial size of the stack allocated for threads created by the server.

Parameters

hServer

The server handle.

Return Value

If the function succeeds, the return value is the amount of memory that will be allocated for the stack in bytes. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpGetServerStackSize** function returns the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `csstools10.h`

Import Library: `cshtsv10.lib`

See Also

[HttpGetServerMemoryUsage](#), [HttpServerStart](#), [HttpSetServerStackSize](#)

HttpGetServerTransferInfo Function

```
BOOL WINAPI HttpGetServerTransferInfo(  
    HSERVER hServer,  
    UINT nClientId,  
    LPHTTPSERVERTRANSFER lpTransferInfo  
);
```

Return information about the current file transfer.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpTransferInfo

A pointer to an **HTTPSERVERTRANSFER** structure that will contain information about the last file transfer. This parameter cannot be NULL, and the *dwSize* member of the structure must be initialized to specify the structure size prior to calling this function.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **HttpGetServerTransferInfo** function is used to obtain information about the last file transfer that was performed by the server for the specified client. This function is typically called within an event handler to determine how many bytes of data were transferred, the type of file and the full path to the file on the local system.

This function will only return information about a file transfer using the GET or PUT commands, with any other command causing this function to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpReceveRequest](#), [HttpSendResponse](#), [HTTPSERVERTRANSFER](#)

HttpGetServerUuid Function

```
INT WINAPI HttpGetServerUuid(  
    HSERVER hServer,  
    UUID * LpUuid  
);
```

Return the UUID assigned to the specified server.

Parameters

hServer

The server handle.

lpUuid

A pointer to a UUID structure that will contain the server UUID when the function returns. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If either the server handle is invalid or the pointer to the UUID structure is NULL, the function will return a value of zero.

Remarks

The **HttpGetServerUuid** function returns the Universally Unique Identifier (UUID) that has been assigned to the server. The UUID may either be generated by the application and assigned as part of the server configuration, or an ephemeral UUID may be automatically generated when the server is started. To obtain a printable string version of the UUID, use the **HttpGetServerUuidString** function.

There is no corresponding function to change the UUID assigned to an active server. The server UUID is assigned when the server is started, and it must be a unique value that is maintained throughout the lifetime of the server. To change the UUID associated with the server, the server must be stopped using the **HttpServerStop** function and another instance of the server started with the new UUID.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

See Also

[HttpGetServerUuidString](#), [HTTPSERVERCONFIG](#)

HttpGetServerUuidString Function

```
INT WINAPI HttpGetServerUuidString(  
    HSERVER hServer,  
    LPTSTR lpszHostUuid,  
    INT nMaxLength  
);
```

Return the UUID assigned to the server as a printable string.

Parameters

hServer

The server handle.

lpszHostUuid

A pointer to a string buffer that will contain the server UUID, terminated with a null character. It is recommended that this buffer be at least 40 characters in length. This parameter cannot be NULL.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. This value must be greater than zero. If the buffer size is too small, the function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the server handle or the buffer is not large enough to store the complete UUID string, the function will return a value of zero.

Remarks

The **HttpGetServerUuidString** function returns the Universally Unique Identifier (UUID) that has been assigned to the server. The UUID may either be generated by the application and assigned as part of the server configuration, or an ephemeral UUID may be automatically generated when the server is started. To obtain the numeric UUID value, use the **HttpGetServerUuid** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetServerUuid](#), [HTTPSERVERCONFIG](#)

HttpGetVirtualHostId Function

```
UINT WINAPI HttpGetVirtualHostId(  
    HSERVER hServer,  
    LPCTSTR lpszHostName,  
    UINT nHostPort  
);
```

Return the virtual host ID associated with the specified hostname.

Parameters

hServer

The server handle.

lpszHostName

A string that specifies the virtual host name.

nHostPort

An integer value which specifies the port number for the virtual host. This value must be zero or the same value as the original port number that the server was configured to use.

Return Value

If the function succeeds, the return value is the host ID that uniquely identifies the virtual host. If the server handle is invalid, or there is no virtual host with the specified name, the function will return VIRTUAL_HOST_UNKNOWN. If the function fails, the last error code will be updated to indicate the cause of the failure.

Remarks

The **HttpGetVirtualHostId** function is used to obtain the unique virtual host ID that is associated with a specific hostname. This function will match both the primary virtual hostname added using the **HttpAddVirtualHost** function, as well as any aliases that were added using the **HttpAddVirtualHostAlias** function. To obtain the virtual host ID associated with the active client session, use the **HttpGetClientVirtualHostId** function.

The *nHostPort* parameter should always be specified with a value of zero, or the same port number that the server was configured to use. Port-based virtual hosting is currently not supported and this parameter is included for future use.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAddVirtualHost](#), [HttpAddVirtualHostAlias](#), [HttpDeleteVirtualHost](#), [HttpGetClientVirtualHostId](#), [HttpGetVirtualHostName](#)

HttpGetVirtualHostName Function

```
INT WINAPI HttpGetVirtualHostName(  
    HSERVER hServer,  
    UINT nHostId,  
    LPTSTR LpszHostName,  
    INT nMaxLength  
);
```

Return the hostname associated with the specified virtual host ID.

Parameters

hServer

The server handle.

nHostId

An integer value which identifies the virtual host.

lpszHostName

A pointer to a string buffer that will contain the virtual host name, terminated with a null character. It is recommended that this buffer be at least 64 characters in length. If this parameter is NULL, the function will return the length of the virtual hostname.

nMaxLength

An integer value that specifies the maximum number of characters can be copied into the string buffer, including the terminating null character. If the *lpszHostName* parameter is NULL this value must be zero.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer, not including the terminating null character. If either the server handle or the host ID is invalid, or the buffer is not large enough to store the complete hostname, the function will return a value of zero.

Remarks

The **HttpGetVirtualHostName** function returns the primary hostname associated with the specified virtual host ID. This is the same hostname that was specified when the virtual host was added to the server configuration using the **HttpAddVirtualHost** function. To obtain the hostname that was used by the active client session to connect to the server, use the **HttpGetClientVirtualHost** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAddVirtualHost](#), [HttpAddVirtualHostAlias](#), [HttpDeleteVirtualHost](#), [HttpGetClientVirtualHost](#), [HttpGetVirtualHostId](#)

HttpIsClientAuthenticated Function

```
BOOL WINAPI HttpIsClientAuthenticated(  
    HSERVER hServer,  
    UINT nClientId  
);
```

Determine if the specified client session has been authenticated.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the client session has been authenticated, this function will return a non-zero value, otherwise it will return zero. If the server handle and client ID are valid, this function will clear the last error code.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[HttpAuthenticateClient](#), [HttpGetClientCredentials](#), [HttpRequireAuthentication](#)

HttpIsCommandEnabled Function

```
BOOL WINAPI HttpIsCommandEnabled(  
    HSERVER hServer,  
    LPCTSTR LpszCommand  
);
```

Determine if a specific server command has been enabled or disabled.

Parameters

hServer

The server handle.

lpszCommand

A pointer to a NULL terminated string that specifies the name of the command. The command name is not case-sensitive, but the value must otherwise match the exact command name.

Partial matches are not recognized by this function. This parameter cannot be NULL.

Return Value

If the command is enabled, this function will return a non-zero value. If the command is disabled, the server handle is invalid or the command name does not match a supported command, this function will return zero.

Remarks

The **HttpIsCommandEnabled** function is used to determine whether a specific command is enabled. Typically this function is used in an event handler to make sure the command issued by a client is recognized by the server and enabled for use. Commands can be enabled and disabled using the **HttpEnableCommand** function.

This function does not account for the permissions granted to a specific client session. Clients are assigned access rights when they are authenticated using the **HttpAuthenticateClient** function, and certain commands can be limited by the permissions granted to the client. For example, even if the PUT command is enabled, a client must have the HTTP_ACCESS_WRITE permission to use the command to upload a file to the server. For a list of access rights, see [User Access Constants](#).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAuthenticateClient](#), [HttpEnableCommand](#), [HttpGetCommandName](#)

HttpReceiveRequest Function

```
BOOL WINAPI HttpReceiveRequest(  
    HSERVER hServer,  
    UINT nClientId,  
    DWORD dwOptions,  
    LPHTTPREQUEST lpRequest,  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength  
);
```

Receive the request that was sent by the client to the server.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

dwOptions

An unsigned integer which specifies how the client request data will be stored. It may be one of the following values:

Constant	Description
HTTP_REQUEST_DEFAULT (0)	If the <i>lpdwLength</i> parameter points to a integer with a non-zero value, the <i>lpvBuffer</i> parameter is considered to be a pointer to a block of memory that has been allocated to store the request data. This is the same as specifying the HTTP_REQUEST_MEMORY option. If the <i>lpdwLength</i> parameter points to an integer with a value of zero, the <i>lpvBuffer</i> parameter is considered to be a pointer to an HGLOBAL memory handle. This is the same as specifying the HTTP_REQUEST_HGLOBAL option.
HTTP_REQUEST_MEMORY (0x1)	The <i>lpvBuffer</i> parameter is a pointer to a block of memory that has been allocated to store the request data. The maximum number of bytes of data that can stored is determined by the value of the integer that the <i>lpdwLength</i> parameter points to. When the function returns, that value will updated with with actual number of bytes copied into the buffer.
HTTP_REQUEST_STRING (0x2)	The <i>lpvBuffer</i> parameter is a pointer to a string buffer that has been allocated to store the request data. The maximum number of bytes of data that can be copied is determined by the value of the integer that the <i>lpdwLength</i> parameter points to. When the function returns, that value will updated with with actual number of bytes copied into the buffer. If the Unicode version of this function is called, the request data will be converted to a null-terminated Unicode string.

HTTP_REQUEST_HGLOBAL (0x4)	The <i>lpvBuffer</i> parameter is a pointer to an HGLOBAL memory handle. When the function returns, the handle will reference a block of memory that contains the request data submitted by the client. The <i>lpdwLength</i> parameter will contain the number of bytes copied to the buffer.
HTTP_REQUEST_FILE (0x8)	The <i>lpvBuffer</i> parameter is a pointer to a string which specifies the name of a file that will contain the request data. If the file does not exist, it will be created. If it does exist, the contents will be replaced. This option is typically used in conjunction with the PUT command. If the <i>lpdwLength</i> parameter is not NULL, the value it points to will be updated with the actual number of bytes stored in the file.
HTTP_REQUEST_HANDLE (0x10)	The <i>lpvBuffer</i> parameter is a handle to an open file. This option is typically used in conjunction with the POST or PUT commands. If the <i>lpdwLength</i> parameter is not NULL, the value it points to will be updated with the actual number of bytes written to the file. If this option is specified, the request data will be written from the current position in the file and will advance the file pointer by the number of bytes received from the client.

lpRequest

A pointer to a **HTTPREQUEST** structure which contains information about the request from the client. This parameter cannot be NULL. The structure that is passed to this function must have all members set to a value of zero except the *dwSize* member, which must be initialized to the size of the structure.

lpvBuffer

A pointer to the buffer that will contain any request data that was submitted by the client. The *dwOptions* parameter determines if this pointer references a block of memory, a null-terminated string buffer, a global memory handle or a file name. If this parameter is NULL, any data submitted by the client will not be copied.

lpdwLength

A pointer to an unsigned integer that will contain the number of bytes of data submitted by the client when the function returns. If the *lpvBuffer* parameter specifies a memory or string buffer, this value must be initialized to the maximum size of the buffer before the function is called. If the *lpvBuffer* parameter points to a global memory handle, this value must be initialized to zero. If *lpvBuffer* is NULL or specifies a file name, this parameter may be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpReceiveRequest** function is called within a HTTP_CLIENT_COMMAND event handler to process the command issued by the client and return information about the request to the server application. It is only necessary for the application to call this function if it wants to implement its own custom handling for a command. It is recommended that most applications use the default command processing for standard commands such as GET and POST to ensure that the

appropriate security checks are performed and the response conforms to the protocol standard.

This function may only be called once per command issued by the client and the data referenced in the **HTTPREQUEST** structure only remains valid while the client is connected to the server. You should never attempt to directly modify the data referenced by any of the structure members. If you wish to store or modify any of the string values returned in the structure, you should allocate a buffer large enough to store the contents of the string, including the terminating null character, and copy the string into that buffer.

If the HTTP_REQUEST_HGLOBAL option is used to return a copy of the request data in a global memory buffer, the HGLOBAL handle must be freed by the application when the data is no longer needed. Failure to free this handle will result in a memory leak.

If the HTTP_REQUEST_HANDLE option is used to write a copy of the request data to an open file, the handle must reference a disk file that was opened or created using the **CreateFile** function with GENERIC_WRITE access. It cannot be a handle to a device or named pipe. If the function succeeds, the file pointer is advanced by the number of bytes of request data submitted by the client. If the function fails, the file pointer is returned to its original position prior to the function being called.

Example

```
// Initialize the HTTPREQUEST structure
HTTPREQUEST httpRequest;
ZeroMemory(&httpRequest, sizeof(httpRequest));
httpRequest.dwSize = sizeof(httpRequest);

// Return the data in a global memory buffer
HGLOBAL hglobalBuffer = NULL;
DWORD dwLength = 0;

bSuccess = HttpReceiveRequest(hServer,
                             nClientId,
                             HTTP_REQUEST_HGLOBAL,
                             &httpRequest,
                             &hglobalBuffer,
                             &dwLength);

if (bSuccess && hglobalBuffer != NULL)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hglobalBuffer);

    if (lpBuffer != NULL)
    {
        // Process dwLength bytes of data submitted by the client
    }

    GlobalUnlock(hglobalBuffer);
    GlobalFree(hglobalBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpSendResponse](#), [HTTPREQUEST](#)

HttpRedirectRequest Function

```
BOOL WINAPI HttpRedirectRequest(  
    HSERVER hServer,  
    UINT nClientId,  
    UINT nMethod,  
    LPCTSTR lpszLocation  
);
```

Redirect the request from the client to another URL.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

nMethod

An integer value that specifies if the redirection is permanent or temporary. The following values may be used:

Constant	Description
HTTP_REDIRECT_PERMANENT (1)	This value is used for permanent redirection, indicating that the client should update any record of the link with the new URL specified by the <i>lpszLocation</i> parameter. This result is cacheable and when the client makes subsequent requests for the resource, it should always use the new URL.
HTTP_REDIRECT_TEMPORARY (2)	This value is used for temporary redirection, indicating that the client should issue a request for the resource using the new URL specified by the <i>lpszLocation</i> parameter, but subsequent requests should continue to use the original URL.
HTTP_REDIRECT_OTHER (3)	This value is used for temporary redirection, however it instructs the client that it should use the GET command to request the redirected resource. This option is typically used to redirect a client after it has used the POST command.

lpszLocation

A pointer to a string that specifies the new location for the requested resource. This value must be a complete URL, including the http:// or https:// scheme. This parameter cannot be NULL or point to zero-length string.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpRedirectRequest** function can be used within a HTTP_CLIENT_COMMAND event

handler to redirect the client to a new location for the resource that it has requested. This redirection can be permanent or temporary, depending on whether the server expects the client to continue to use the original URL when requesting the resource.

If the `HTTP_REDIRECT_TEMPORARY` method is used, the actual status code that is returned to the client depends on the version of the protocol that is being used. If the client has issued the request using HTTP 1.0 then the server will return a 302 code to the client. If the client used HTTP 1.1, the server will return a 307 code to the client that indicates it should use the same command verb (GET, POST, etc.) when requesting the resource at the new location.

If the `HTTP_REDIRECT_OTHER` method is used, the status code that is returned to the client depends on which version of the protocol is being used. For clients who are using HTTP 1.0, the server will return a 302 code to the client just as with the `HTTP_REDIRECT_TEMPORARY` method. If the client is using HTTP 1.1, the server will return a 303 code to the client that indicates it should always use the GET command to request the new resource, regardless if a different command was originally used (POST, PUT, etc.)

This function provides a simplified interface for sending a redirection status code that also implicitly sets the Location response header to the value of the *lpzLocation* parameter. If the server application needs to send alternate redirection codes such as 305 (Use Proxy) then it should use the **HttpSendReponse** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshtsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpRequireAuthentication](#), [HttpSendErrorResponse](#), [HttpSendResponse](#)

HttpRegisterHandler Function

```
BOOL WINAPI HttpRegisterHandler(  
    HSERVER hServer,  
    UINT nHostId,  
    LPCTSTR lpszExtension,  
    LPCTSTR lpszProgramFile,  
    LPCTSTR lpszParameters,  
    LPCTSTR lpszDirectory  
);
```

Register a CGI program for use and associate it with a file name extension.

Parameters

hServer

The server handle.

nHostId

An unsigned integer that identifies the virtual host associated with the program. The value VIRTUAL_HOST_DEFAULT should be used for the default host that is created when the server is first started.

lpszExtension

A pointer to a string which specifies the file name extension that is associated with the CGI program. This parameter cannot be NULL.

lpszProgramFile

A pointer to a string that specifies the full path to the executable program. This parameter cannot be NULL.

lpszParameters

A pointer to a string that specifies additional parameters for the program. This value will be passed to the program as command line arguments. If the program does not require any command line parameters, this value may be NULL or point to an empty string.

lpszDirectory

A pointer to a string that specifies the current working directory for the program. If this parameter is NULL or points to an empty string, the server will use the root document directory for the virtual host.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpRegisterHandler** function registers an executable CGI program and associates it with a file name extension. When the client issues a GET or POST command that specifies a file with that extension, the program will be executed and the output return to the client.

The *lpszProgramFile* string specifies file name of the CGI program. You should not install any executable programs in the server root directory or its subdirectories. A client should never have the ability to directly access the executable file itself. It is permitted to have multiple file name extensions that reference the same program. The only requirement is that the extension be unique for the given host. The program name may contain environment variables surrounded by %

symbols. For example, %ProgramFiles% would be expanded to the **C:\Program Files** folder.

It is important to note that the program specified by *lpszProgramFile* must be an executable file, not a script or batch file. If the program name does not contain a directory path, then the standard Windows pathing rules will be used when searching for an executable file that matches the given name. It is recommended that you always provide a full path to the executable file.

The *lpszParameters* string can specify additional command line parameters that should be passed to the CGI program as arguments. This string can also contain a placeholder named "%1" that will be replaced by the full path to the local script filename. If no placeholder is included in the parameters, or *lpszParameters* is a NULL pointer, the script file name will be passed to the program as its only argument.

The executable program that is registered using this program must be a console application that conforms to the CGI/1.1 specification defined in RFC 3875. Request data submitted by the client as part of a POST will be provided to the program as standard input. The output from the program must be written to standard output. The first lines of output from the program should be any response headers, followed by an empty line. Each line should be terminated with a carriage-return and linefeed (CRLF) sequence. If the CGI program outputs additional data to be processed by the client, it should provide Content-Type and Content-Length response headers.

The application can obtain a copy of the output from the command by calling the **HttpGetProgramOutput** function from within a HTTP_CLIENT_EXECUTE event handler.

When developing a CGI program, it is important to take into consideration the environment that it will be executing in. The program will be started as a child process of the server application, and will inherit the same privileges. This means that it will typically have access to the boot drive, the Windows folders and the system registry. CGI programs must ensure that all query parameters and request data submitted by the client have been validated.

If the server is running on a system with User Account Control (UAC) enabled and does not have elevated privileges, do not register a program that requires elevated privileges or has a manifest that specifies the requestedExecutionLevel as requiring administrative privileges.

Example

```
// Register a handler for VBScript
HttpRegisterHandler(hServer,
    VIRTUAL_HOST_DEFAULT,
    _T("vbs"),
    _T("%SystemRoot%\System32\cscript.exe"),
    _T("/nologo /b \"%1\""),
    NULL);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetProgramExitCode](#), [HttpGetProgramOutput](#), [HttpRegisterProgram](#)

HttpRegisterProgram Function

```
BOOL WINAPI HttpRegisterProgram(  
    HSERVER hServer,  
    UINT nHostId,  
    LPCTSTR lpszVirtualPath,  
    LPCTSTR lpszProgramFile,  
    LPCTSTR lpszParameters,  
    LPCTSTR lpszDirectory  
);
```

Register a CGI program for use and associate it with a virtual path on the server.

Parameters

hServer

The server handle.

nHostId

An unsigned integer that identifies the virtual host associated with the program. The value VIRTUAL_HOST_DEFAULT should be used for the default host that is created when the server is first started.

lpszVirtualPath

A pointer to a string which specifies the virtual path to the CGI program. This must be an absolute path, but does not have to specify a pre-existing virtual path or map to the directory structure of the root document directory for the server. This parameter cannot be NULL. The maximum length of the virtual path is 1024 characters.

lpszProgramFile

A pointer to a string that specifies the full path to the executable program. This parameter cannot be NULL.

lpszParameters

A pointer to a string that specifies additional parameters for the program. This value will be passed to the program as command line arguments. If the program does not require any command line parameters, this value may be NULL or point to an empty string.

lpszDirectory

A pointer to a string that specifies the current working directory for the program. If this parameter is NULL or points to an empty string, the server will use the root document directory for the virtual host.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpRegisterProgram** function registers an executable CGI program and associates it with a virtual path. When the client issues a GET or POST command specifying the virtual path associated with the program, the program will be executed and the output return to the client.

The *lpszProgramFile* string specifies file name of the CGI program. You should not install any executable programs in the server root directory or its subdirectories. A client should never have the ability to directly access the executable file itself. It is permitted to have multiple virtual paths

that reference the same executable file. The only requirement is that the virtual path be unique for the given host. The program name may contain environment variables surrounded by % symbols. For example, %ProgramFiles% would be expanded to the **C:\Program Files** folder.

It is important to note that the program specified by *lpszProgramFile* must be an executable file, not a script or batch file. If the program name does not contain a directory path, then the standard Windows pathing rules will be used when searching for an executable file that matches the given name. It is recommended that you always provide a full path to the executable file.

The *lpszParameters* string can specify additional command line parameters that should be passed to the CGI program as arguments. This string can also contain a placeholder named "%1" that will be replaced by the virtual path associated with the program. If *lpszParameters* is NULL or a zero-length string, then no additional parameters are passed to the program.

The executable program that is registered using this program must be a console application that conforms to the CGI/1.1 specification defined in RFC 3875. Request data submitted by the client as part of a POST will be provided to the program as standard input. The output from the program must be written to standard output. The first lines of output from the program should be any response headers, followed by an empty line. Each line should be terminated with a carriage-return and linefeed (CRLF) sequence. If the CGI program outputs additional data to be processed by the client, it should provide Content-Type and Content-Length response headers.

The application can obtain a copy of the output from the command by calling the **HttpGetProgramOutput** function from within a HTTP_CLIENT_EXECUTE event handler.

When developing a CGI program, it is important to take into consideration the environment that it will be executing in. The program will be started as a child process of the server application, and will inherit the same privileges. This means that it will typically have access to the boot drive, the Windows folders and the system registry. CGI programs must ensure that all query parameters and request data submitted by the client have been validated.

If the server is running on a system with User Account Control (UAC) enabled and does not have elevated privileges, do not register a program that requires elevated privileges or has a manifest that specifies the requestedExecutionLevel as requiring administrative privileges.

Example

```
HttpRegisterProgram(hServer,  
                   VIRTUAL_HOST_DEFAULT,  
                   HTTP_METHOD_DEFAULT,  
                   _T("/order/invoice"),  
                   _T("%ProgramData%\MyServer\Programs\invoice.exe"),  
                   NULL,  
                   NULL);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetProgramExitCode](#), [HttpGetProgramOutput](#), [HttpRegisterHandler](#)

HttpRenameServerLogFile Function

```
BOOL WINAPI HttpRenameServerLogFile(  
    HSERVER hServer,  
    LPCTSTR lpszFileName  
);
```

Rename or delete the current log file being updated by the server.

Parameters

hServer

The server handle.

lpszFileName

A pointer to a string that specifies the file name the current log file should be renamed to. If this parameter is NULL or an empty string, the current log file will be deleted.

Return Value

If the function succeeds, the return value is non-zero. If the server handle does not specify a valid server, the function will return zero. If logging is not currently enabled for the server, this function will return zero.

Remarks

The **HttpRenameServerLogFile** function is used to rename or delete the current log file. Note that this does not change the current log file name or disable logging by the server. It only changes the file name of the current log file, or removes the log file if the *lpszFileName* parameter is NULL. This can be useful if you want your server to perform log file rotation, archiving the current log file. By renaming the current log file, the server will automatically create a new log file with original file name.

This function must be used to rename or delete the current log file while logging is active because the server holds an open handle on the file. The application should not use the **HttpGetServerLogFile** function to obtain the log file name and then use the **MoveFileEx** or **DeleteFile** functions with that file.

To disable logging, use the **HttpSetServerLogFile** function and specify the logging format as HTTP_LOGFILE_NONE.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetServerLogFile](#), [HttpSetServerLogFile](#)

HttpRequireAuthentication Function

```
BOOL WINAPI HttpRequireAuthentication(  
    HSERVER hServer,  
    UINT nClientId,  
    UINT nAuthType,  
    LPCTSTR lpszRealm  
);
```

Send a response to the client indicating that authentication is required.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

nAuthType

An integer value that corresponds to a result code, informing the client if the redirection is permanent or temporary. The following values may be used:

Constant	Description
HTTP_AUTH_BASIC (1)	This option specifies the Basic authentication scheme should be used. This option is supported by all clients that support at least version 1.0 of the protocol.

lpszRealm

A pointer to a string that is displayed by a web browser to indicate to the user which username and password they should use. If this parameter is NULL or an empty string, the domain name the client used to establish the connection will be used.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpRequireAuthentication** function can be used within a HTTP_CLIENT_COMMAND event handler to indicate to the client that it must provide a username and password to access the requested resource. The client should respond by issuing another request that includes the required credentials. To determine if a client has included credentials with its request, use the **HttpIsClientAuthenticated** function. The **HttpGetClientCredentials** function will return the username and password that was provided by the client.

Some clients may require that the session be secure if authentication is requested or display warning messages to the user if the connection is not secure. It is recommended that you enable security using the HTTP_OPTION_SECURE option if your application will require clients to authenticate before accessing specific resources.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetClientCredentials](#), [HttpIsClientAuthenticated](#), [HttpSendResponse](#)

HttpSendErrorResponse Function

```
BOOL WINAPI HttpSendErrorResponse(  
    HSERVER hServer,  
    UINT nClientId,  
    UINT nErrorCode,  
    LPCTSTR lpszMessage  
);
```

Send a customized error response to the specified client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

nErrorCode

An integer value that specifies the error code that should be sent to the client. This value should correspond to the error result codes defined for HTTP in RFC 2616, which are three-digit values in the range of 400 through 599. The function will fail if an invalid error code is specified.

lpszMessage

A pointer to a string that describes the error. If this parameter is NULL or specifies a zero-length string, a default message will be selected based on the error code.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpSendErrorResponse** function sends a response to the client indicating that an error has occurred, providing a numeric error code and HTML formatted text which may be displayed to the user. The *lpszMessage* parameter should provide a brief description of the error that will be included in the output sent to the client. Note that the message should not contain any special formatting control characters or HTML markup.

This function provides a simplified interface for sending an error response to the client. In some cases, a browser may choose to display its own error message to the user in place of the generic HTML document generated by this function. If you want your application to send a customized HTML document for a specific type of error, you should use the **HttpSendResponse** function.

If you wish to redirect the client to use an alternate URL to access the requested resource, it is recommended that you use the **HttpRedirectRequest** function rather than sending an error response.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpRequest](#), [HttpSendResponse](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

HttpSendResponse Function

```
BOOL WINAPI HttpSendResponse(  
    HSERVER hServer,  
    UINT nClientId,  
    DWORD dwOptions,  
    LPHTTPRESPONSE LpResponse,  
    LPVOID lpvBuffer,  
    DWORD dwLength  
);
```

Send a response from the server to the specified client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

dwOptions

An unsigned integer which specifies how the client request data will be copied. It may be one of the following values:

Constant	Description
HTTP_RESPONSE_DEFAULT (0)	The <i>lpvBuffer</i> parameter is a pointer to a block of memory that contains the response data. The number of bytes of data that in the buffer is specified by the <i>dwLength</i> parameter. This is the same as specifying the HTTP_RESPONSE_MEMORY option.
HTTP_RESPONSE_MEMORY (0x1)	The <i>lpvBuffer</i> parameter is a pointer to a block of memory that contains the response data. The number of bytes of data that in the buffer is specified by the <i>dwLength</i> parameter. The data will be sent to the client as a stream of bytes. If the server application was compiled using Unicode, it is responsibility of the application to convert any Unicode text to either ANSI or UTF-8, depending on the resource that was requested by the client.
HTTP_RESPONSE_STRING (0x2)	The <i>lpvBuffer</i> parameter is a pointer to a string buffer that contains the response data. The maximum number of bytes of data that will be sent to the client is determined by the <i>dwLength</i> parameter. If the value of the <i>dwLength</i> parameter exceeds the string length, the value will be ignored and the contents of the string will be sent up to the terminating null character. If the Unicode version of this function is called, the string will be converted to a byte array before being sent to the client.
HTTP_RESPONSE_HGLOBAL	The <i>lpvBuffer</i> parameter is an HGLOBAL memory

(0x4)	handle which references a block of memory that contains the response data. The number of bytes of data that in the buffer is specified by the dwLength parameter. The data will be sent to the client as a stream of bytes. It is the responsibility of the application to free the global memory handle after it is no longer needed.
HTTP_RESPONSE_FILE (0x8)	The lpvBuffer parameter is a pointer to a string which specifies the name of a file that contains the response data. If the file does not exist, or does not specify a regular file, this function will fail. The dwLength parameter is ignored. If the content type for the specified file is not explicitly defined in the response, the function will attempt to automatically determine the correct type based on the file name extension and/or the contents of the file.
HTTP_RESPONSE_HANDLE (0x10)	The lpvBuffer parameter is a handle to an open file and the dwLength parameter specifies the number of bytes to be read from the file and send to the client. If this option is specified, the response data will be read from the current position in the file and will advance the file pointer by the number of bytes sent to the client.
HTTP_RESPONSE_DYNAMIC (0x100)	The response data will be generated dynamically. This prevents the content length from being included in the response header, and forces the connection to close, regardless if the client has requested to keep the connection open. This option must be specified if the application wishes to use the HttpSendResponseData function to send additional data to the client.
HTTP_RESPONSE_NOCACHE (0x200)	Informs the client that the data being returned by the server should not be cached. Typically this is used in conjunction with the HTTP_RESPONSE_DYNAMIC option when the data is being generated dynamically.

lpResponse

A pointer to a **HTTPRESPONSE** structure which contains additional information about the response to the client. The structure that is passed by reference to this function must have the **dwSize** member initialized to the size of the structure or the function will fail. This parameter may be NULL, in which case a default response of "200 OK" is sent to the client along with any data specified by the **lpvBuffer** parameter.

lpvBuffer

A pointer to the buffer that will contain any response data that should be sent to the client. The **dwOptions** parameter determines if this pointer references a block of memory, a null-terminated string buffer, a global memory handle or a file name. This parameter may be NULL, in which case no data will be sent to the client. If this function is called in response to a HEAD command being sent by the client, this parameter is ignored.

dwLength

An unsigned integer that specifies the number of bytes of data to be sent to the client. If the *lpvBuffer* parameter is NULL, this value must be zero. If this function is called in response to a HEAD command being sent by the client, this parameter is ignored.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpSendResponse** function is called within a HTTP_CLIENT_COMMAND event handler to respond to the request made by the client. This function may only be called once after a command has been received, and must be called after the **HttpReceiveRequest** function. It is only necessary for the application to call this function if it wants to implement its own custom handling for a command. It is recommended that most applications use the default command processing for standard commands such as GET and POST to ensure that the appropriate security checks are performed and the response conforms to the protocol standard.

If the HTTP_RESPONSE_HANDLE option is used to read a copy of the response data from an open file, the handle must reference a disk file that was opened using the **CreateFile** function with GENERIC_READ access. It cannot be a handle to a device or named pipe. If the *dwLength* parameter is larger than the total number of bytes available to be read from the current position in the file, the function will stop sending data to the client when it reaches the end-of-file. If the function succeeds, the file pointer is advanced by the number of bytes of response data sent to the the client. If the function fails, the file pointer is returned to its original position prior to the function being called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpRecieveRequest](#), [HttpSendResponseData](#), [HTTPRESPONSE](#)

HttpSendResponseData Function

```
INT WINAPI HttpSendResponseData(  
    HSERVER hServer,  
    UINT nClientId,  
    LPBYTE lpBuffer,  
    DWORD dwLength,  
    LPDWORD lpdwResponse  
);
```

Send additional data to the client in response to a command.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpBuffer

A pointer to a buffer that contains the response data that should be sent to the client. This parameter cannot be NULL and must point to a valid byte buffer. If the response contains non-ASCII text characters, it should be UTF-8 encoded.

dwLength

An unsigned integer that specifies the size of the response data sent to the client. This value must be greater than zero and should never exceed the number of bytes stored in the buffer specified by the *lpBuffer* parameter.

lpdwResponse

A pointer to an unsigned integer which will contain the total number of bytes successfully sent to the client in response to the request. This parameter may be NULL, in which case it is ignored.

Return Value

If the function succeeds, the return value is the number of bytes of data sent to the client. If the function fails, the return value is HTTP_ERROR. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpSendResponseData** function is called within a HTTP_CLIENT_COMMAND event handler to send data to the client in response to a request. This function can only be used to send dynamically generated content after the **HttpSendResponse** function has been called. The HTTP_RESPONSE_DYNAMIC option must have been specified when responding to the client or this function will fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpRecieveRequest](#), [HttpSendResponse](#), [HTTPRESPONSE](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

HttpServerAsyncNotify Function

```
BOOL WINAPI HttpServerAsyncNotify(  
    HSERVER hServer,  
    HWND hWnd,  
    UINT uMsg  
);
```

Enable or disable asynchronous notification of changes in server status.

Parameters

hServer

The server handle.

hWnd

A handle to the window whose window procedure will receive the notification message.

uMsg

The user-defined message that will be sent to the notification window.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call the **HttpGetServerError** function.

Remarks

The **HttpServerAsyncNotify** function is used by an application to enable or disable asynchronous notifications. The message window is typically the main UI window and these notifications are used signal to the application that it should update the user interface. If the *hWnd* parameter is not NULL, it must specify a valid window handle and the user-defined message must have a value of **WM_USER** or higher. The application cannot specify a notification message that is reserved by the operating system. The pseudo-handle **HWND_BROADCAST** cannot be specified as the notification window. If the *hWnd* parameter is NULL, notifications for the specified server will be disabled.

When asynchronous notifications are enabled for a server, the server will post the user-defined message to the window whenever there is a change in status or after a client has connected or disconnected from the server. The *wParam* message parameter will contain the notification message and the *lParam* message parameter will contain the handle to the server or the client ID. The following notification messages are defined:

Constant	Description
HTTP_NOTIFY_STARTUP	This notification is sent when the server has started and is preparing to accept client connections. This notification is only sent once, and only if asynchronous notifications are enabled immediately after the HttpServerStart function is called. This message will not be sent once the server has begun accepting client connections or when notification messages are disabled and then subsequently re-enabled at a later time. The <i>lParam</i> message parameter will specify the handle to the server.
HTTP_NOTIFY_LISTEN	This notification is sent when the server is listening for

	<p>client connections. This notification message may be sent to the application multiple times over the lifetime of the server. If the server was suspended, this notification will be sent after the application calls the HttpServerResume function to resume accepting client connections. The <i>lParam</i> message parameter will specify the handle to the server.</p>
HTTP_NOTIFY_SUSPEND	<p>This notification is sent when the server suspends accepting new connections because the application has called the HttpServerSuspend function. This notification message may be sent to the application multiple times over the lifetime of the server. The <i>lParam</i> message parameter will specify the handle to the server.</p>
HTTP_NOTIFY_RESTART	<p>This notification is sent when the server is restarted using the HttpServerRestart function. Note that the server socket handle provided by the <i>lParam</i> message parameter will specify the new socket handle of the restarted server instance, not the original socket handle. The <i>lParam</i> message parameter will specify the handle to the server.</p>
HTTP_NOTIFY_CONNECT	<p>This notification is sent when the server accepts a client connection and the thread that manages the client session has begun processing network events for that client. This message notification will not be sent if the client connection is rejected by the server. The <i>lParam</i> message parameter will specify the unique ID of the client that connected to the server.</p>
HTTP_NOTIFY_DISCONNECT	<p>This notification is sent when the client disconnects from the server and the client socket has been closed. This notification message may not occur for each client session that is forced to terminate as the result of the server being stopped using the HttpServerStop function. The <i>lParam</i> message parameter will specify the unique ID of the client that disconnected from the server.</p>
HTTP_NOTIFY_SHUTDOWN	<p>This notification is sent when the server thread is in the process of terminating. At the time the application processes this notification message, the server handle in <i>lParam</i> will reference the defunct server and cannot be used with other server functions. The <i>lParam</i> message parameter will specify the handle to the server.</p>

If asynchronous notifications are enabled, you should never use those notifications as a replacement for an event handler. When an event occurs, the callback function that handles the event is invoked in the context of the thread that manages the client session. The application should exchange data with the client within that event handler and not in response to a notification message. These notification messages should only be used to update the application

UI in response to changes in the status of the server.

The HTTP_NOTIFY_CONNECT and HTTP_NOTIFY_DISCONNECT notifications are different from the other server notifications because the *lParam* message parameter does not specify the server handle, but rather the unique client ID associated with the session that connected to or disconnected from the server. Use the **HttpGetClientServer** function to obtain a handle to the server that created the client session. Note that at the time the application processes the HTTP_NOTIFY_DISCONNECT notification message, the client session will have already terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include ctools10.h

Import Library: cshtsv10.lib

See Also

[HttpGetClientServer](#), [HttpServerStart](#)

HttpServerDisableTrace Function

```
BOOL WINAPI HttpServerDisableTrace();
```

Disable the logging of network function calls.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[HttpServerEnableTrace](#)

HttpServerEnableTrace Function

```
BOOL WINAPI HttpServerEnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

Enable the logging of network function calls to a file.

Parameters

lpszTraceFile

A pointer to a string that specifies the name of the log file. If this parameter is NULL or points to an empty string, a log file is created in the temporary directory for the current user.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_DEFAULT (0)	All function calls are written to the trace file. This is the default value.
TRACE_ERROR (1)	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING (2)	Only those function calls which fail, or return values which indicate a warning, are recorded in the trace file.
TRACE_HEXDUMP (4)	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call [HttpGetServerError](#).

Remarks

When trace logging is enabled, the log file is opened, appended to and closed for each socket function call. Using the same file name, you can do the same in your application to add additional information to the file if needed. This can provide an application-level context for the entries made by the library. Make sure that the file is closed after the data has been written. If a file name is not specified by the caller, a file named **cstrace.log** will be created in the temporary directory for the current user.

The TRACE_HEXDUMP option can produce very large files, since all data that is being sent and received by the application is logged. To reduce the size of the file, you can enable and disable logging around limited sections of code that you wish to analyze.

To redistribute an application that includes this debug logging functionality, the **cstrcv10.dll** library must be included as part of the installation package. This library provides the trace logging features, and if it is not available the **HttpServerEnableTrace** function will fail. Note that this is a standard Windows DLL and does not need to be registered, it only needs to be redistributed with your application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpServerDisableTrace](#)

HttpServerInitialize Function

```
BOOL WINAPI HttpServerInitialize(  
    LPCTSTR LpszLicenseKey,  
    LPINITDATA LpData  
);
```

The **HttpServerInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

LpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

LpData

Pointer to an INITDATA data structure. This parameter may be NULL if the initialization data for the library is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**. All other functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must insure that this function is called with a valid license key.

If the *LpData* argument is specified, it must point to an INITDATA structure which has been initialized by setting the *dwSize* member to the size of the structure. All other structure members should be set to zero. If the function is successful, then the INITDATA structure will be filled with identifying information about the library.

Although it is only required that **HttpServerInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **HttpServerUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **HttpServerInitialize** or **HttpServerUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpServerStart](#), [HttpServerStop](#), [HttpServerUninitialize](#), [INITDATA](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

HttpServerProc Function

```
VOID CALLBACK HttpServerProc(  
    HSERVER hServer,  
    UINT nClientId,  
    UINT nEventId,  
    DWORD dwError,  
    DWORD_PTR dwParam  
);
```

The **HttpServerProc** function is an application-defined callback function that processes events generated by the server process.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client that has issued a request to the server.

nEventId

An unsigned integer which specifies which event occurred. For a list of events, see [Server Event Constants](#).

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero. The **HttpGetServerError** function can be used to obtain additional information about the error code.

dwParam

A user-defined integer value which was specified when the server was started. This value is guaranteed to be large enough to store a pointer or handle value on both 32-bit and 64-bit platforms.

Return Value

None.

Remarks

The HTTP_CLIENT_COMMAND event can be used to filter commands issued by the client. If the command performs an operation on a file or directory, the **HttpGetCommandFile** function can be used to obtain the file name on the server that the client specified. It is possible for the application to change the file name using the **HttpSetCommandFile** function to direct the server to use a different file than the one specified by the client. If the file name is changed and the event handler returns a value of zero, the server will perform the default action for the command using the new file name.

The callback function will be called in the context of the thread that is currently managing the client session. You must ensure that any access to global or static variables are synchronized, otherwise the results may be unpredictable. It is recommended that you do not declare any static variables within the callback function itself.

If the application has a graphical user interface, you should never attempt to directly modify a UI control from within an event handler. Controls should only be modified by the same UI thread that created their window. To change the user interface in response to a server event, use the

HttpServerAsyncNotify function to enable asynchronous notifications and update the UI in response to the notification message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[HttpServerAsyncNotify](#), [HttpServerStart](#)

HttpServerRestart Function

```
HSERVER WINAPI HttpServerRestart(  
    HSERVER hServer  
);
```

Restart the server, terminating all active client sessions.

Parameters

hServer

The server handle.

Return Value

If the function succeeds, the return value is the new handle for the specified server. If the function fails, the return value is INVALID_SERVER. To get extended error information, call

HttpGetServerError.

Remarks

The **HttpServerRestart** function will restart the specified server, terminating all active client sessions. The server handle that is returned by the function is the handle for the new server instance, and the old handle value is no longer valid. If the function is unable to restart the server for any reason, the server thread is terminated. The server retains all of the configuration parameters from the previous instance, however the statistical information (such as the number of clients, files transferred, etc.) will be reset.

If an application calls this function from within an event handler, the active client session (the client for which the event handler was invoked) may not get a disconnect notification. It is recommended that this function only be called by the same thread that created the server using the **HttpServerStart** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

See Also

[HttpServerStart](#), [HttpServerStop](#)

HttpServerResume Function

```
BOOL WINAPI HttpServerResume(  
    HSERVER hServer  
);
```

Resume accepting client connections.

Parameters

hServer

Handle to the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpServerResume** function instructs the server to resume accepting new client connections after the **HttpServerSuspend** function has been called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

See Also

[HttpServerRestart](#), [HttpServerStart](#), [HttpServerStop](#), [HttpServerSuspend](#), [HttpServerThrottle](#)

HttpServerStart Function

```
HSERVER WINAPI HttpServerStart(  
    LPCTSTR lpszLocalHost,  
    UINT nLocalPort,  
    DWORD dwOptions,  
    LPHTTPSERVERCONFIG lpServerConfig,  
    HTTPSETPROCBASE lpServerProc,  
    DWORD_PTR dwServerParam,  
    LPSECURITYCREDENTIALS lpCredentials  
);
```

The **HttpServerStart** function begins listening for client connections on the specified local address and port number. The server is started in its own thread and manages the client sessions independently of the calling thread. All interaction with the server and its client sessions takes place inside the callback function specified by the caller.

Parameters

lpszLocalHost

A pointer to a string which specifies the local hostname or IP address address that the server should be bound to. If this parameter is NULL or an empty string, then an appropriate address will automatically be used. If a specific address is used, the server will only accept client connections on the network interface that is bound to that address.

nLocalPort

The port number the server should use to accept client connections. If a value of zero is specified, the server will use the standard port number 21 to listen for connections, or port 990 if the server is configured to use implicit SSL. The port number used by the application must be unique and multiple instances of a server cannot use the same port number. It is recommended that a port number greater than 5000 be used for private, application-specific implementations.

dwOptions

An unsigned integer value that specifies the options used when creating an instance of the server. For a list of options, see [Server Option Constants](#).

lpServerConfig

A pointer to an **HTTPSERVERCONFIG** structure that specifies the configuration options for the server. If this parameter is NULL, then default configuration parameters will be used that starts the server in a restrictive mode.

lpServerProc

Specifies the address of the application defined callback function. For more information about the callback function, see the description of the **HttpServerProc** callback function. If this parameter is NULL, a default internal handler is used to process client commands.

dwServerParam

A user-defined integer value that is passed to the callback function. If the *lpServerProc* parameter is NULL, this value should be zero. This value is guaranteed to be large enough to store a pointer or handle value on both 32-bit and 64-bit platforms.

lpCredentials

Pointer to a **SECURITYCREDENTIALS** structure. If this parameter is NULL, the default security credentials for the server host name will be used. If security is enabled for the server, it is recommended that you provide a pointer to this structure with specific information about the

server certificate that should be used. If the security options are not specified in the server configuration, this parameter is ignored.

Return Value

If the function succeeds, the return value is a handle to a server session. If the function fails, the return value is `INVALID_SERVER`. To get extended error information, call **HttpGetServerError**.

Remarks

In most cases, the *lpzLocalHost* parameter should be a NULL pointer or an empty string. On a multihomed system, this will enable the server to accept connections on any appropriately configured network adapter. Specifying a hostname or IP address will limit client connections to that particular address. Note that the hostname or address must be one that is assigned to the local system, otherwise an error will occur.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the function will fail.

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

The handle returned by this function references the listening socket that was created when the server was started. The service is managed in another thread, and all interaction with the server and active client connections are performed inside the event handler. To disconnect all active connections, close the listening socket and terminate the server thread, call the **HttpServerStop** function.

The host UUID that is defined as part of the server configuration should be generated using the **uuidgen** utility that is included with the Windows SDK. You should not use the UUID that is provided in the example code, it is for demonstration purposes only. If no host UUID is specified in the server configuration, an ephemeral UUID will be generated automatically when the server is started.

If the server is started with security enabled, the *lpCredentials* parameter should be used to provide the server certificate information in a **SECURITYCREDENTIALS** structure. If the pointer is NULL, this function will search for a certificate that matches the hostname provided as part of the server configuration. This requires that the server certificate be installed in the personal certificate store of the current process user, and it must have a private key associated with it. To use a certificate with a different name, or one that is stored in a PFX file, a **SECURITYCREDENTIALS** structure must be defined and passed to this function.

Example

```
HSERVER hServer;  
HTTPSERVERCONFIG httpConfig;  
  
// Initialize the server configuration  
ZeroMemory(&httpConfig, sizeof(HTTPSERVERCONFIG));  
httpConfig.dwSize = sizeof(httpConfig);  
httpConfig.nMaxClients = 100;  
httpConfig.nMaxClientsPerAddress = 8;  
httpConfig.nMaxRequests = 5;  
httpConfig.nLogFormat = HTTP_LOGFILE_EXTENDED;
```

```
httpConfig.nLogLevel = 5;
httpConfig.lpszIdentity = _T("MyProgram");
httpConfig.lpszHostName = _T("server.company.com");
httpConfig.lpszHostUuid = _T("10000000-1000-1000-1000-100000000000");
httpConfig.lpszDirectory = _T("%ProgramData%\MyProgram\Files");
httpConfig.lpszLogFile = _T("%ProgramData%\MyProgram\Server.log");

// Start the server
hServer = HttpServerStart(lpszLocalHost,
                        HTTP_PORT_DEFAULT,
                        HTTP_SERVER_DEFAULT,
                        &httpConfig,
                        lpEventHandler,
                        0,
                        NULL);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpEnumServerClients](#), [HttpServerProc](#), [HttpServerRestart](#), [HttpServerStop](#), [HTTPSERVERCONFIG](#), [SECURITYCREDENTIALS](#)

HttpServerStop Function

```
BOOL WINAPI HttpServerStop(  
    HSERVER hServer  
);
```

Stop the server, terminating all active client sessions and releasing the resources that were allocated for the server.

Parameters

hServer

The server handle.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid, the function will return a value of zero.

Remarks

The **HttpServerStop** function instructs the server to stop accepting client connections, disconnects all active client connections and terminates the thread that is managing the server session. The handle is no longer valid after the server has been stopped and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new server is started. An application should always consider the server handle to be opaque and never depend on it being a specific value.

If an application calls this function from within an event handler, the active client session (the client for which the event handler was invoked) may not get a disconnect notification. It is recommended that this function only be called by the same thread that created the server using the **HttpServerStart** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cshtsv10.lib`

See Also

[HttpServerRestart](#), [HttpServerStart](#)

HttpServerSuspend Function

```
BOOL WINAPI HttpServerSuspend(  
    HSERVER hServer  
);
```

Suspend the server and reject new client connections.

Parameters

hServer

Handle to the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpServerSuspend** function instructs the server to suspend accepting new client connections. Any incoming client connections will be rejected with an error message indicating that the server is currently unavailable. To resume accepting client connections, call the **HttpServerResume** function. Suspending the server will have no effect on clients that have already established a connection with the server.

It is recommended that you only suspend a server if absolutely necessary, and only for brief periods of time. If you want to limit the number of active client connections or control the connection rate for clients, use the **HttpServerThrottle** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

See Also

[HttpServerRestart](#), [HttpServerResume](#), [HttpServerStart](#), [HttpServerStop](#), [HttpServerThrottle](#)

HttpServerThrottle Function

```
BOOL WINAPI HttpServerThrottle(  
    HSERVER hServer,  
    UINT nMaxClients,  
    UINT nMaxClientsPerAddress,  
    DWORD dwConnectionRate  
);
```

The **HttpServerThrottle** function limits the number of active client connections, connections per address and connection rate.

Parameters

hServer

Handle to the server.

nMaxClients

A value which specifies the maximum number of clients that may connect to the server. A value of zero specifies that there is no fixed limit to the number of client connections.

nMaxClientsPerAddress

A value which specifies the maximum number of clients that may connect to the server from the same IP address. A value of zero specifies that there is no fixed limit to the number of client connections per address. By default, there is a limit of four client connections per address.

dwConnectionRate

A value which specifies a restriction on the rate of client connections, limiting the number of connections that will be accepted within that period of time. A value of zero specifies that there is no restriction on the rate of client connections. The higher this value, the fewer the number of connections that will be accepted within a specific period of time. By default, there is no limit on the client connection rate.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpServerThrottle** function is used to limit the number of connections and the connection rate to minimize the potential impact of a large number of client connections over a short period of time. This can be used to protect the server from a client application that is malfunctioning or a deliberate denial-of-service attack in which the attacker attempts to flood the server with connection attempts.

If the maximum number of client connections or maximum number of connections per address is exceeded, the server will reject subsequent connection attempts until the number of active client sessions drops below the specified threshold. Note that adjusting these values lower than the current connection limits will not affect clients that have already connected to the server. For example, if the **HttpServerStart** function is called with the maximum number of clients set to 100, and then **HttpServerThrottle** is called lowering that value to 75, no existing client connections will be affected by the change. However, the server will not accept any new connections until the number of active clients drops below 75.

Increasing the connection rate value will force the server to slow down the rate at which it will

accept incoming client connection requests. For example, setting this parameter to a value of 1000 would limit the server to accepting one client connection every second, while a value of 250 would allow the server to accept four client connections per second. Note that significantly increasing the amount of time the server must wait to accept client connections can exceed the connection backlog queue, resulting in client connections being rejected.

It is recommended that you always specify conservative connection limits for your server application based on expected usage. Allowing an unlimited number of client connections can potentially expose the system to denial-of-service attacks and should never be done for servers that are accessible over the Internet.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

See Also

[HttpServerRestart](#), [HttpServerResume](#), [HttpServerStart](#), [HttpServerSuspend](#)

HttpServerUninitialize Function

```
VOID WINAPI HttpServerUninitialize();
```

The **HttpServerUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **HttpServerInitialize** call before it can call any of the other library functions. When it has completed the use of library, the application must call **HttpServerUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There must be a call to **HttpServerUninitialize** for every successful call to **HttpServerInitialize** made by a process. Operations for all threads in the server are terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpServerInitialize](#), [HttpServerStart](#), [HttpServerStop](#)

HttpSetClientAccess Function

```
BOOL WINAPI HttpSetClientAccess(  
    HSERVER hServer,  
    UINT nClientId,  
    DWORD dwUserAccess  
);
```

Change the access rights associated with the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

dwUserAccess

An unsigned integer which specifies one or more user access rights. For a list of user access rights that can be granted to the client, see [User and File Access Constants](#).

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero. This function can only be used with authenticated clients. If the client session has not been authenticated, the return value will be zero.

Remarks

The **HttpSetClientAccess** function can change the multiple access rights for a client session. The **HttpEnableClientAccess** function can be used to grant or revoke a specific permission for the client session.

If the *dwUserAccess* parameter has a value of HTTP_ACCESS_DEFAULT, then default permissions will be granted to the client session based on the configuration of the server. This is the recommended value for most clients. It is important to consider the implications of changing the access permissions granted to a client session. For example, if you do not grant clients HTTP_ACCESS_READ permission, it can effectively disable the site because the server will return 403 Forbidden errors for all GET and HEAD requests.

This function should typically be called in the HTTP_CLIENT_CONNECT event handler to assign general permissions to the client, or in the HTTP_CLIENT_COMMAND event handler after the client has issued a request and provided any authentication credentials that are required.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

See Also

[HttpAuthenticateClient](#), [HttpEnableClientAccess](#), [HttpGetClientAccess](#)

HttpSetClientHeader Function

```
BOOL WINAPI HttpSetClientHeader(  
    HSERVER hServer,  
    UINT nClientId,  
    UINT nHeaderType,  
    LPCTSTR lpszHeaderName,  
    LPCTSTR lpszHeaderValue  
);
```

Create or change the value of a request or response header for the client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

nHeaderType

Specifies the type of header to create or modify. It may be one of the following values:

Constant	Description
HTTP_HEADERS_REQUEST	Change a request header that was provided by the client. Request header values provide additional information to the server about the type of request being made.
HTTP_HEADERS_RESPONSE	Change a response header that was created by the server. Response header values provide additional information to the client about the type of information that is being returned by the server.

lpszHeaderName

A pointer to a string that specifies the name of the header that should be created or modified. Header names are not case-sensitive and should not include the colon which acts as a delimiter that separates the header name from its value. This parameter cannot be a NULL pointer or an empty string.

lpszHeaderValue

A pointer to a string that specifies the new value of the header. If this parameter is a NULL pointer or an empty string, it has the same effect as deleting the header value from the list of request or response headers.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero. If the function fails, the **HttpGetServerError** function will return more information about the last error that has occurred.

Remarks

The **HttpSetClientHeader** function will change the value of a request or response header for the specified client session. If the *lpszHeaderName* value matches an existing header field, its value will be replaced. If the header name is not defined, then a new header will be created with the

given value. You should not change the value of most standard response header values unless you are certain of the impact that it would have on the normal operation of the client.

If you wish to define a custom header value that would be included in the response to a client request, you should prefix the header name with "X-" to avoid potential conflicts with the standard response headers. For example, if you wanted to identify a customer, you could create a header field with the name "X-Customer-ID" and set the value to the customer ID number. The client application would receive this custom header value as part of the response to its request for a document.

Refer to [Hypertext Transfer Protocol Headers](#) for a list of common request and response headers that are used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpDeleteClientHeader](#), [HttpGetClientHeader](#)

HttpSetClientIdleTime Function

```
UINT WINAPI HttpSetClientIdleTime(  
    HSERVER hServer,  
    UINT nClientId,  
    UINT nTimeout  
);
```

Change the idle timeout period for the specified client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

nTimeout

An unsigned integer value that specifies the number of seconds that the client may remain idle. If this value is zero, the default idle timeout period for the server will be used.

Return Value

If the function succeeds, the return value is the previous client idle timeout period in seconds. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **HttpSetClientIdleTime** function will change the number of seconds that the client may remain idle before being automatically disconnected by the server. The minimum timeout period for a client is 10 seconds, the maximum is 300 seconds (5 minutes). The idle time of a client session is based on the last time a command was issued to the server or when a data transfer completed.

If the value INFINITE is specified as the timeout period, the client activity timer will be refreshed, extending the idle timeout period for the session. This is typically done inside an event handler to prevent the client from being disconnected due to inactivity.

To obtain the current idle timeout period for a client, along with the amount of time the client has been idle, use the **HttpGetClientIdleTime** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

See Also

[HttpGetClientIdleTime](#)

HttpSetClientVariable Function

```
BOOL WINAPI HttpSetClientVariable(  
    HSERVER hServer,  
    UINT nClientId,  
    LPCTSTR lpszName,  
    LPCTSTR lpszValue  
);
```

Create or change the value of a CGI environment variable for the specified client.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszName

A pointer to a string that specifies the name of the environment variable that should be created or modified. Environment variables are not case-sensitive and should not include the equal sign which acts as a delimiter that separates the variable name from its value. This parameter cannot be a NULL pointer or an empty string.

lpszValue

A pointer to a string that specifies the new value of the environment variable. If this parameter is a NULL pointer or an empty string, it has the same effect as deleting the variable from the environment block for the client session.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero. If the function fails, the **HttpGetServerError** function will return more information about the last error that has occurred.

Remarks

The **HttpSetClientVariable** function will change the value of a environment variable for the specified client session. If the *lpszName* value matches an existing variable, its value will be replaced. If the variable is not defined, then a new variable will be created with the given value. The value of an environment variable can be obtained using the **HttpGetClientVariable** function.

The server will automatically create a number of different environment variables that will be passed to a program or script executed by the server. These variables are defined in RFC 3875 as part of the Common Gateway Interface (CGI) 1.1 specification. The following variables are defined by the server and should not be modified directly by the application:

Variable Name	Description
AUTH_TYPE	The authorization scheme used by the server to authenticate the client session
CONTENT_LENGTH	The length of the request data provided by the client
CONTENT_TYPE	The MIME type that identifies the type of content provided by the client

DOCUMENT_ROOT	The full path to the local document root directory on the server
GATEWAY_INTERFACE	The version of the Common Gateway Interface that is being used by the server
PATH_INFO	The resource or sub-resource that is to be returned by the program or script
PATH_TRANSLATED	The path information mapped to the server root document directory structure
QUERY_STRING	The URL encoded query parameters passed to the program or script
REMOTE_ADDR	The network address of the client sending the request to the server
REMOTE_HOST	The same value as the REMOTE_ADDR variable
REMOTE_USER	The username specified as part of the authentication credentials provided by the client
REQUEST_METHOD	The method used by the client to request the resource
REQUEST_URI	The URI for the script provided by the client
SCRIPT_FILENAME	The full path to the program or script on the server
SCRIPT_NAME	The path to the program or script specified by the client
SERVER_NAME	The hostname or IP address of the server that the client connected to
SERVER_PORT	The port number that the client used to connect to the server
SERVER_PORT_SECURE	This variable has a value of "1" if the client connection to the server is secure
SERVER_PROTOCOL	The version of the server protocol used
SERVER_SOFTWARE	The server identity string which specifies the application name and version

In addition to the environment variables listed, the server will also create variables that are prefixed with "HTTP_" that are set to the value of request headers that are not otherwise defined. For example, the HTTP_USER_AGENT variable will be set to the value of the User-Agent header provided by the client as part of the request.

Calling the **HttpSetClientVariable** function in response to an HTTP_CLIENT_EXECUTE event notification will have no effect because it occurs after the CGI program or script has completed execution. To create or modify environment variables for the client session, it should be done in response to the HTTP_CLIENT_COMMAND event notification.

This function will not change the environment block for the server process. Each client session is allocated its own private environment block which is inherited by the CGI program. When the client session terminates, the memory allocated for its environment is released.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetClientVariable](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

HttpSetCommandFile Function

```
BOOL WINAPI HttpSetCommandFile(  
    HSERVER hServer,  
    UINT nClientId,  
    LPCTSTR lpszFileName  
);
```

Change the name of the local file or directory that is the target of the current command.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszFileName

A pointer to a string that specifies the new file name. This parameter may be NULL to specify that the original file or directory name should be used.

Return Value

If the function succeeds, the return value is non-zero. If the server handle and client ID do not specify a valid client session, the function will return zero.

Remarks

The **HttpSetCommandFile** function is used by the application to change the target file or directory name for the current command from within an HTTP_EVENT_COMMAND event handler. This can be used to effectively redirect the client to use a different file than the one that was actually requested. For example, if the client issues the GET command to download a file from the server, this function can be used to redirect the command to use a different file name. To obtain the full path to the file or directory that is the target of the current command, use the **HttpGetCommandFile** function.

The *lpszFileName* parameter specifies the path to the new file or directory name. If the path is absolute, then it will be used as-is. If the path is relative, it will be relative to the root directory for the client session. Because the root document directory for the client can depend on the hostname that the client used when connecting to the server, it is recommended that you always use an absolute path. The full path to this file is not limited to the server root directory or its subdirectory, it can specify a file anywhere on the local system. If this parameter is a NULL pointer, or points to an empty string, then the server will revert to using the actual file or directory name specified by the command. This enables the application to effectively undo a previous call to this function to change the target file name.

Typically this function would be used to redirect a client to a file or directory that it may not normally have access to. Exercise caution when using this function to provide access to data that is stored outside of the server root directory. Incorrect use of this function could expose the server to security risks or cause unpredictable behavior by client applications. In most cases it is preferable to use the **HttpAddVirtualPath** function to create a virtual path or file name on the server, or the **HttpRedirectRequest** function to request the client use a different URL to access the resource.

This function should only be called within the context of the HTTP_EVENT_COMMAND event, and only for those commands that perform an action on a file or directory. If the current command

does not target a file or directory, this function will return zero and the last error code will be set to ST_ERROR_INVALID_COMMAND.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpAddVirtualPath](#), [HttpGetCommandFile](#), [HttpReceiveRequest](#), [HttpRedirectRequest](#),
[HttpSendResponse](#)

HttpSetServerError Function

```
VOID WINAPI HttpSetServerError(  
    HSERVER hServer,  
    DWORD dwError  
);
```

Set the last error code for the specified server session.

Parameters

hServer

The server handle.

dwError

An unsigned integer that specifies an error code.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each server session. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SERVER or HTTP_ERROR.

If the *hServer* parameter is specified as INVALID_SERVER, this function will set the last error code for the current thread, but will not change the error code associated with any server session. This should only be done if the application does not have access to a valid server handle.

If the *dwError* parameter is specified with a value of zero, this effectively clears the error code for the last function that failed. Those functions which clear the last error code when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **HttpGetServerError** function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

See Also

[HttpGetCommandResult](#), [HttpGetServerError](#), [HttpSendResponse](#)

HttpSetServerIdentity Function

```
BOOL WINAPI HttpSetServerIdentity(  
    HSERVER hServer,  
    LPCTSTR lpszIdentity  
);
```

Change the identity of the specified server.

Parameters

hServer

The server handle.

lpszIdentity

A pointer to a string that identifies the server. If this parameter is NULL or specifies an empty string, the current identity for the server is reset to a default value. The maximum length of the identity string is 64 characters, including the terminating null character.

Return Value

If the function succeeds, the return value is non-zero. If the server handle is invalid, the function will return a value of zero.

Remarks

The **HttpSetClientIdentity** function changes a string value used by the server to identify itself to clients. The identity string does not have any standard format and is used for informational purposes only. Typically it consists of the application name and a version number. Changing the server identity has no effect on the operation of the server. To obtain the identity string currently associated with the server, use the **HttpGetServerIdentity** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetServerIdentity](#)

HttpSetServerLogFile Function

```
BOOL WINAPI HttpSetServerLogFile(  
    HSERVER hServer,  
    UINT nLogFormat,  
    UINT nLogLevel,  
    LPCTSTR lpszFileName  
);
```

Change the current log format, level of detail and file name.

Parameters

hServer

The server handle.

nLogFormat

An integer value that specifies the format used when creating or updating the server log file. The following formats are supported:

Constant	Description
HTTP_LOGFILE_NONE (0)	This value specifies that the server should not create or update a log file.
HTTP_LOGFILE_COMMON (1)	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about GET, PUT and POST requests.
HTTP_LOGFILE_COMBINED (2)	This value specifies that the server should use the combined log file format. This format is similar to the common format, however it includes the client referrer and user agent. This is the format that most Apache web servers use by default.
HTTP_LOGFILE_EXTENDED (3)	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session. This format typically generates the largest logfiles.

nLogLevel

An integer value that specifies the level of detail that should be generated in the log file. The minimum value is 1 and the maximum value is 10. If this parameter is zero, it is the same as specifying a log file format of HTTP_LOGFILE_NONE and will disable logging by the server.

lpszFileName

A pointer to a string that specifies the name of the log file that should be created or appended to. If the server was configured with logging enabled and this parameter is NULL or an empty string, the current log file name will not be changed. If the log file does not exist, it will be created. If it does exist, the contents of the log file will be appended to.

Return Value

If the function succeeds, the return value is non-zero. If the server handle does not specify a valid

server, the function will return zero.

Remarks

The **HttpSetServerLogFile** function can be used to change the current log file name, the format of the log file or the level of detail recorded in the log file. In some situations it may be desirable to delete the current log file contents when changing the format or ensure that a new log file is created. To do this, combine the *nLogFormat* parameter with the constant HTTP_LOGFILE_DELETE.

The higher the value of the *nLogLevel* parameter, the greater the level of detail that is recorded by the server. A log level of 1 instructs the server to only record GET, POST and PUT requests, while a level of 10 instructs the server to record all commands processed by the server. Because a higher level of logging detail can negatively impact the performance of the server, it is recommended that you do not exceed a level of 5 for most applications. A log level of 10 should only be used for debugging purposes.

Example

```
UINT nLogFormat = HTTP_LOGFILE_NONE;
UINT nLogLevel = 0;
UINT nNewLevel = 5;
BOOL bChanged = FALSE;

// Change the level of detail for the current log file if logging
// has been enabled and the current level is a lower value

if (HttpGetServerLogFile(hServer, &nLogFormat, &nLogLevel, NULL, 0))
{
    if (nLogFormat != HTTP_LOGFILE_NONE && nLogLevel < nNewLevel)
        bChanged = HttpSetServerLogFile(hServer, nLogFormat, nNewLevel, NULL);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoos10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetServerLogFile](#), [HttpRenameServerLogFile](#)

HttpSetServerName Function

```
BOOL WINAPI HttpSetServerName(  
    HSERVER hServer,  
    UINT nClientId,  
    LPCTSTR lpszHostName  
);
```

Change the host name assigned to the specified server or client session.

Parameters

hServer

The server handle.

nClientId

An unsigned integer which uniquely identifies the client session. This value may be zero.

lpszHostName

A pointer to a string that specifies the new host name assigned to the server or client session. If this value is NULL or points to an empty string, the current host name will be changed to use the default host name.

Return Value

If the function succeeds, the return value is non-zero. If either the server handle or the client ID is invalid, or the buffer is not large enough to store the complete hostname, the function will return a value of zero.

Remarks

This function will change the host name assigned to the specified client session. If the *nClientId* parameter has a value of zero, the function will change default host name that was assigned to the server as part of the server configuration. If the *nClientId* parameter specifies a valid client session and the *lpszHostName* parameter is NULL, the host name associated with the client session will be changed to the current host name assigned to the server.

When a client connects to the server, it can specify the host name that it used to establish the connection by sending the HOST command. This is typically used with virtual hosting, where one server may accept client connections for multiple domains. The **HttpGetServerName** function will return the host name specified by the client, and **HttpSetServerName** can be used by the application to either explicitly assign a different host name to the client session, or override the host name provided by the client.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cshtsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetServerAddress](#), [HttpGetServerName](#)

HttpSetServerPriority Function

```
INT WINAPI HttpSetServerPriority(  
    HSERVER hServer,  
    INT nPriority  
);
```

Change the priority assigned to the specified server.

Parameters

hServer

The server handle.

nPriority

An integer value which specifies the new priority for the server. It may be one of the following values:

Constant	Description
HTTP_PRIORITY_BACKGROUND (0)	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
HTTP_PRIORITY_LOW (1)	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
HTTP_PRIORITY_NORMAL (2)	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
HTTP_PRIORITY_HIGH (3)	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
HTTP_PRIORITY_CRITICAL (4)	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Return Value

If the function succeeds, the return value is the previous priority assigned to the server. If the function fails, the return value is HTTP_PRIORITY_INVALID. To get extended error information, call

HttpGetServerError.

Remarks

The **HttpSetServerPriority** function can be used to change the current priority assigned to the specified server. Client connections that are accepted after this function is called will inherit the new priority as their default priority. Previously existing client connections will not be affected by this function.

Higher priority values increase the thread priority and processor utilization for each client session. You should only change the server priority if you understand the impact it will have on the system and have thoroughly tested your application. Configuring the server to run with a higher priority can have a negative effect on the performance of other programs running on the system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cstools10.h`

Import Library: `cshtsv10.lib`

See Also

[HttpGetServerPriority](#), [HttpServerStart](#)

HttpSetServerStackSize Function

```
BOOL WINAPI HttpSetServerStackSize(  
    SOCKET hServer,  
    DWORD dwStackSize  
);
```

Change the initial size of the stack allocated for threads created by the server.

Parameters

hServer

Handle to the server socket.

dwStackSize

The amount of memory that will be committed to the stack for each thread created by the server. If this value is zero, a default stack size will be used.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **HttpGetServerError**.

Remarks

The **HttpSetServerStackSize** function changes the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes. Increasing or decreasing the stack size will only affect new threads that are created by the server, it will not affect those threads that have already been created to manage active client sessions. It is recommended that most applications use the default stack size.

You should not change the stack size unless you understand the impact that it will have on your system and have thoroughly tested your application. Increasing the initial commit size of the stack will remove pages from the total system commit limit, and every page of memory that is reserved for stack cannot be used for any other purpose.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `csstools10.h`

Import Library: `cshtsv10.lib`

See Also

[HttpGetServerMemoryUsage](#), [HttpGetServerStackSize](#), [HttpServerStart](#)

Hypertext Transfer Protocol Server Data Structures

- HTTPCLIENTCREDENTIALS
- HTTPREQUEST
- HTTPRESPONSE
- HTTPSERVERCONFIG
- HTTPSERVERTRANSFER
- INITDATA
- SECURITYCREDENTIALS

HTTPCLIENTCREDENTIALS Structure

The **HTTPCLIENTCREDENTIALS** structure defines the credentials used to authenticate a specific user.

```
typedef struct _HTTPCLIENTCREDENTIALS
{
    DWORD dwSize;
    DWORD dwFlags;
    UINT nAuthType;
    UINT nHostPort;
    TCHAR szHostName[HTTP_MAXHOSTNAME];
    TCHAR szUserName[HTTP_MAXUSERNAME];
    TCHAR szPassword[HTTP_MAXPASSWORD];
} HTTPCLIENTCREDENTIALS, *LPHTTPCLIENTCREDENTIALS;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwFlags

An unsigned integer value is reserved for future use. This value will always be zero.

nAuthType

An unsigned integer value that identifies the authentication method.

nHostPort

The server port number.

szHostName

A pointer to a string that specifies the server host name.

szUserName

A pointer to a string that specifies the user name.

szPassword

A pointer to a string that specifies the user password.

Remarks

When an instance of this structure is passed to the **HttpGetClientCredentials** function, this member must be initialized to the size of the structure and all other members must be initialized with a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetClientCredentials](#)

HTTPREQUEST Structure

The **HTTPREQUEST** structure provides information about the request made by the client.

```
typedef struct _HTTPREQUEST
{
    DWORD    dwSize;
    DWORD    dwFlags;
    DWORD    dwVersion;
    DWORD    dwAccess;
    DWORD    dwLength;
    DWORD    dwReserved;
    UINT     nHostId;
    UINT     nHostPort;
    LPCSTR   lpszCommand;
    LPCSTR   lpszHostName;
    LPCSTR   lpszUserName;
    LPCSTR   lpszPassword;
    LPCSTR   lpszResource;
    LPCSTR   lpszParameters;
    LPCSTR   lpszDirectory;
    LPCSTR   lpszPathInfo;
    LPCSTR   lpszProgram;
    LPCSTR   lpszScriptFile;
    LPCSTR   lpszLocalFile;
    LPCSTR   lpszMediaType;
} HTTPREQUEST, *LPHTTPREQUEST;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwFlags

An unsigned integer value that provides additional information about the request. It may be one or more of the following values:

Constant	Description
HTTP_REQUEST_FLAG_PROTECTED (0x1)	The resource that the client has requested is protected. The client should only be permitted to access the resource if the client session has been authenticated.
HTTP_REQUEST_FLAG_PROGRAM (0x2)	The resource that the client has requested is a CGI program or an executable script. The output from the program or script should be returned to the client in the response. If this flag is specified, the <i>lpszProgram</i> member of the structure specifies the name of the program that should be executed.
HTTP_REQUEST_FLAG_SCRIPT (0x4)	The resource that the client has requested is an executable script. This flag is set when the request URL is mapped to a registered script handler. The <i>lpszProgram</i> member specifies the name of the program that is responsible for executing the

script, and the *lpszScriptFile* parameter specifies the full path to the script itself.

dwVersion

An unsigned integer value that specifies the protocol version used. It may be one of the following values:

Constant	Description
HTTP_VERSION_09 (0x00009)	The client issued a GET command without specifying a protocol version.
HTTP_VERSION_10 (0x10000)	The client issued a command that specified version 1.0 of the protocol.
HTTP_VERSION_11 (0x10001)	The client issued a command that specified version 1.1 of the protocol.

dwAccess

An unsigned integer value that specifies the access permissions that were assigned to the resource. For a list of file access permissions, see [User and File Access Constants](#).

dwLength

An unsigned integer value that will specify the number of bytes of request data provided by the client. If the client did not submit any data with the request, this member will have a value of zero.

dwReserved

An unsigned integer value that is reserved for future use.

nHostId

An integer value that identifies the virtual host that was specified by the client. This is based on the value of the Host request header included in the request. This value will be zero if a host name is not specified by the client, or does not match one of the virtual hosts that have been created.

nHostPort

An integer value that specifies the port number that the client used to establish the connection. This will be the same port number that was used when starting the server.

lpszCommand

A pointer to a string that specifies the command that was issued by the client. The command will always be in upper case. Refer to [Hypertext Transfer Protocol Commands](#) for a list of standard commands.

lpszHostName

A pointer to a string that identifies the hostname or IP address that the client used to establish the connection. This will typically correspond to the hostname assigned to the server, or to one of the virtual hosts that have been created.

lpszUserName

A pointer to a string that specifies the username provided with the request. This member is only set if the client has included authentication credentials as part of the request. If the client has not provided any credentials, this member will be NULL.

lpszPassword

A pointer to a string that specifies the password provided with the request. This member is only set if the client has included authentication credentials as part of the request. If the client has not provided any credentials, this member will be NULL.

lpszResource

A pointer to a string that specifies the URL path provided by the client.

lpszParameters

A pointer to a string that specifies any query parameters that were included in the request made by the client. If the URL did not include any query parameters, this member will be NULL.

lpszDirectory

A pointer to a string that specifies the full path to the root directory for the virtual host.

lpszPathInfo

A pointer to a string that specifies additional path information provided by the client when referencing an executable CGI program or script in the URL. For example, if a program is mapped to the virtual path "/orders/invoice" and the client requests "/orders/invoice/pdf/10001" this member will be the string "/pdf/10001". If there is no path information associated with the request, this member will be NULL.

lpszProgram

A pointer to a string that specifies the local path to the CGI program that should be executed to process the request made by the client. If this member is NULL, it indicates that the client has provided a URL that maps to a local file or directory.

lpszScriptFile

A pointer to a string that specifies the local path to the script file that will be executed by the handler. If this member is NULL, it indicates that the request URL was not mapped to a registered script handler.

lpszLocalFile

A pointer to a string that specifies the local path to the directory or file name referenced by the URL. If the *lpszPathInfo* member is not NULL, then this member will point to a local file name based on the path information appended to the root directory for the virtual host.

lpszMediaType

A pointer to a string that identifies the request data using the standard Internet media types defined in RFC 2046. It is used to designate the format for various types of content and has two parts, a primary and secondary media type separated by a forward slash. Common examples are "text/plain", "text/html" and "application/octet-stream". If the client did not submit any data with the request, this member will be NULL.

Remarks

This structure is used with the **HttpReceiveRequest** function and all members should be initialized to a value of zero, except for the *dwSize* member which should be initialized to the size of the structure. Failure to properly initialize the structure will cause the function call to fail.

The value of the *lpszLocalFile* member depends on whether the client has requested a static document, or if the URL is mapped to a registered program or script handler. If the request is for a static document, then the *dwFlags* member will not have the HTTP_REQUEST_FLAG_PROGRAM bit flag set and the *lpszLocalFile* member will be the full path to the document. If the URL is mapped to a registered program or script, then the *lpszLocalFile* member will be the full path to the server root directory. If the request URL included additional path information, that will be appended to the root directory.

The value of the *lpzScriptFile* member is only defined if the *dwFlags* member has the HTTP_REQUEST_FLAG_SCRIPT bit flag set. This indicates that the request URL references a file that is an executable script with a handler that was registered using the **HttpRegisterHandler** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpReceiveRequest](#), [HttpSendResponse](#), [HTTPRESPONSE](#)

HTTPRESPONSE Structure

The **HTTPRESPONSE** structure provides additional information about the response to the client.

```
typedef struct _HTTPRESPONSE
{
    DWORD    dwSize;
    DWORD    dwFlags;
    DWORD    dwVersion;
    DWORD    dwReserved;
    UINT     nExpires;
    UINT     nResultCode;
    LPCTSTR  lpszReason;
    LPCTSTR  lpszMediaType;
} HTTPRESPONSE, *LPHTTPRESPONSE;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwFlags

An unsigned integer value that specifies one or more response flags. This member is reserved for future use and should always have a value of zero.

dwVersion

An unsigned integer value that specifies the protocol version used. It may be one of the following values:

Constant	Description
HTTP_VERSION_DEFAULT (0)	The server should respond to the client using the same version specified in the request.
HTTP_VERISON_10 (0x10000)	The server should respond to the client using version 1.0 of the protocol.
HTTP_VERSION_11 (0x10001)	The server should respond to the client using version 1.1 of the protocol.

dwReserved

An unsigned integer value that is reserved for future use.

nExpires

An integer value that specifies the number of seconds until the client should consider any cached response data to be stale. If this value is zero, the default cache expiration time for the server will be used. The default cache expiration time is 7200 seconds (2 hours). The value of this structure member is ignored if the **HTTP_RESPONSE_NOCACHE** option is specified when the response is sent to the client.

nResultCode

An integer value that specifies the result code that should be sent in response to the client request. The result code is a three-digit number that indicates success or failure. For more information, refer to the **HttpGetCommandResult** function.

lpszReason

A string that describes the result code sent to the client. The description should be brief

and should not contain any formatting characters or HTML markup. This parameter may be NULL, in which case a default description of the result code will be used.

lpzMediaType

A string that specifies the Internet media type for the data that is being sent to the client as defined in RFC 2046. The format for the content type string consists of two parts, a primary and secondary media type separated by a forward slash. Common examples are "text/plain", "text/html" and "application/octet-stream". If this member is NULL, the library will attempt to automatically determine the appropriate media type.

Remarks

This structure is used with the **HttpSendResponse** function and the *dwSize* member must be initialized to size of the structure. Failure to properly initialize the structure will cause the function call to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetCommandResult](#), [HttpReceiveRequest](#), [HttpSendResponse](#), [HTTPREQUEST](#)

HTTPSERVERCONFIG Structure

The **HTTPSERVERCONFIG** structure provides the configuration information used to create an instance of an HTTP server.

```
typedef struct _HTTPSERVERCONFIG
{
    DWORD    dwSize;
    UINT     nMaxClients;
    UINT     nMaxClientsPerAddress;
    UINT     nMaxRequests;
    UINT     nMaxPostSize;
    UINT     nLogFormat;
    UINT     nLogLevel;
    UINT     nCacheTime;
    UINT     nIdleTime;
    UINT     nExecTime;
    LPCTSTR  lpszIdentity;
    LPCTSTR  lpszHostUuid;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszRootPath;
    LPCTSTR  lpszTempPath;
    LPCTSTR  lpszLogFile;
} HTTPSERVERCONFIG, *LPHTTPSERVERCONFIG;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

nMaxClients

An integer value that specifies the maximum number of active client connections that will be accepted by the server. If the maximum number of clients is reached, any further connections are rejected by the server until one or more clients close their connection to the server or are disconnected. A value of zero specifies the default configuration value of 100 connections should be used.

nMaxClientsPerAddress

An integer value that specifies the maximum number of active client connections per IP address that will be accepted by the server. If the maximum number of clients from the same IP address is reached, any further connections are rejected until one of the clients closes its connection to the server. A value of zero specifies the default configuration value of 8 connections per address should be used.

nMaxRequests

An integer value that specifies the maximum number of a requests a client may make per connection. When a client reaches the maximum number of requests, the server will close the connection and the client must establish a new connection to send another request. If this member has a value of zero, the server will allow 5 requests per connection. The minimum value is 1 and the maximum is 100.

nMaxPostSize

An integer value that specifies the maximum size of the request data that may be submitted by a client with a POST command. If the amount of request data submitted to the server exceeds this limit, the server will return a 413 error to the client and the connection will be closed. If this member has a value of zero, the server will permit the client to submit up to 4 MB (4,194,304

bytes) of data. The minimum value is 100 KB (102,400 bytes) and the maximum value is 100 MB (104,857,600 bytes).

nLogFormat

An integer value that specifies the format used when creating or updating the server log file. The following formats are supported:

Constant	Description
HTTP_LOGFILE_NONE (0)	This value specifies that the server should not create or update a log file.
HTTP_LOGFILE_COMMON (1)	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about file transfers.
HTTP_LOGFILE_COMBINED (2)	This value specifies that the server should use the combined log file format. This format is similar to the common format, however it includes the client referrer and user agent. This is the format that most Apache web servers use by default.
HTTP_LOGFILE_EXTENDED (3)	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session.

nLogLevel

An integer value that specifies the level of detail that should be generated in the log file. The minimum value is 1 and the maximum value is 10. If the *nLogFormat* member specifies a valid log file format and this value is zero, a default level of detail will be selected based on the format. The common log file format generally contains less information by default, only logging the data transfers between the client and server. The W3C extended log file format defaults to a higher level of detail that includes additional information about the client session. The higher the level of detail, the larger the log file will be.

nCacheTime

An integer value that specifies the amount of time that the client should store a document in its cache before requesting it from the server again. A value of zero specifies that the default value of 7200 seconds (2 hours) should be used.

nIdleTime

An integer value that specifies the maximum number of seconds that a client session may be idle before the server closes the control connection to the client. A value of zero specifies the default value of 60 seconds. If the value is non-zero, the minimum value is 10 seconds and the maximum value is 300 seconds (5 minutes). This value is used to initialize the default idle timeout period for each client session. The server determines if a client is idle based on the time the last command was issued and whether or not a file transfer is in progress.

nExecTime

An unsigned integer value that specifies the maximum number of seconds that an external CGI program is permitted to run on the server. Programs are registered using the **HttpRegisterProgram** function, and are executed when the client sends a request for a resource that is associated with the program. If this value is zero, the default timeout period of 5

seconds will be used. The minimum execution time is 1 second and the maximum time limit is 30 seconds.

lpszIdentity

A pointer to a string that identifies the server. It is used for informational purposes only and has no effect on the operation of the server. If this member is not initialized to a value, then a default identity will be used. The server identity is provided when a client establishes the initial connection and when the client sends the STAT or CSID commands. If this member is defined, it is recommended that you only use printable ASCII characters.

lpszHostUuid

A pointer to a string that specifies a Universally Unique Identifier (UUID) that is used to uniquely identify the server. This value can be used when storing information about the server, and should be generated using a utility such as **uuidgen** which is included with Visual Studio. This structure member may be initialized to an empty string, in which case a temporary UUID will be randomly generated.

lpszHostName

A pointer to a string that specifies the fully-qualified host name for the server. If this structure member is initialized with an empty string, the local host name assigned to the Windows system will be used. This value does not need to correspond to the actual host name associated with the server's IP address. This value is used for informational purposes only and has no effect on the operation of the server.

lpszRootPath

A pointer to a string that specifies the path to the root directory for the server. If this structure member is initialized with an empty string, the current working directory for the process will be used as the root directory. It is recommended that you provide a full path to an existing directory and do not specify the root of a local or network drive. If the path does not exist at the time the server is started, it will be automatically created.

lpszTempPath

A pointer to a string that specifies the path to the temporary directory for the server. If this structure member is initialized with an empty string, the current temporary directory for the process will be used. The temporary directory cannot be the same as the root directory, and it is strongly recommended that you do not specify the root of a local or network drive. If the path does not exist at the time the server is started, it will be automatically created.

lpszLogFile

A pointer to a string that specifies the name of the server log file to create, if a logging format has been specified. If logging is enabled and this member is an empty string, then a default log file name will be created. If the file name does not include a path, then the file is created in the server log directory. If the file name includes a path, the log file will be created using that specific name. If the server is in multi-user mode, then the default location for log files will be the Logs subdirectory in the server root directory. If the server is not in multi-user mode, the default location for log files will be the temporary directory for the current process.

Remarks

When an instance of this structure is passed to the **HttpServerStart** function, the **dwSize** member must be initialized to the size of the structure, otherwise the function will fail with an error indicating that the configuration is invalid.

The **nMaxClients** member limits the total number of client connections and the **nMaxClientsPerAddress** member limits the number of simultaneous connections can originate

from the same address. If either client limit is exceeded, the server will automatically close the connection. For more control over how the server accepts client connections, use the **HttpServerThrottle** function. It is not recommended that you set the maximum clients per address below a value of 4. Lower values can negatively impact the performance of some clients, particularly web browsers.

The **nMaxPostSize** member limits the amount of data that a client can submit to the server using a POST command. The default limit is 4 MB which should be sufficient for most applications. One situation where you may need to increase this value is if you expect the client to upload large files using the POST command. Increasing this limit will potentially increase the amount of virtual memory that the server will allocate. Note that this value does not limit amount of data that can be uploaded using the PUT command.

When specifying the root directory for the server configuration, it is important to consider that a client's default access to files on the system will be limited to these directories and any subdirectories. If a root directory is not specified, then one will be created using the current working directory for the process. The root directory path may include environment variables surrounded by % symbols and these will be expanded.

For servers that are publicly accessible, or where you want files to be accessible across multiple server sessions, you should always populate the **szHostUuid** member with a valid UUID string, and the **lpszRootPath** member should specify an absolute path to an existing directory.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpServerStart](#), [HttpServerStop](#), [HttpServerThrottle](#)

HTTPSERVERTRANSFER Structure

The **HTTPSERVERTRANSFER** structure provides information about the last file transfer performed by a client.

```
typedef struct _HTTPSERVERTRANSFER
{
    DWORD          dwSize;
    DWORD          dwReserved;
    DWORD          dwFileAccess;
    DWORD          dwTimeElapsed;
    ULARGE_INTEGER uiBytesCopied;
    TCHAR          szFileName[MAX_PATH];
} HTTPSERVERTRANSFER, *LPHTTPSERVERTRANSFER;
```

Members

dwSize

An unsigned integer value that specifies the size of the structure.

dwReserved

An unsigned integer value that is reserved for future use. This value will always be zero.

dwFileAccess

An unsigned integer value that specifies the how the local file was accessed. It can be one of the following values:

Constant	Description
HTTP_FILE_READ (0)	The file was opened for reading. This mode indicates that the client issued the GET command to download the contents of a file from the server to the client system. The <i>szFileName</i> member specifies the name of the local file on the server that was downloaded by the client.
HTTP_FILE_WRITE (1)	The file was opened for writing. This mode indicates that the client issued the PUT command to upload the contents of a file from the client system to server. The <i>szFileName</i> member specifies the name of the local file on the server that was created by the client. If a file already existed with the name name, it was replaced.

dwTimeElapsed

The amount of time that it took for the file transfer to complete in milliseconds. This value is limited to the resolution of the system timer, which is typically in the range of 10 to 16 milliseconds. This value may be zero if the transfer occurred over a local network or on the same host using a loopback address.

uiBytesCopied

A 64-bit integer value that specifies the total number of bytes copied during the file transfer. This value is represented by a ULARGE_INTEGER union which provides support for those programming languages that do not have intrinsic support for 64-bit integers. For more information, refer to the Windows SDK documentation. The application should not make the assumption that this is the actual size of the file.

szFileName

A pointer to a string value that will contain the full path to the local file that was transferred. The

dwFileAccess member determines whether the file name represents a file that was downloaded by the client, or uploaded from the client and stored on the server.

Remarks

When an instance of this structure is passed to the **HttpGetServerTransferInfo** function, the *dwSize* member must be initialized to the size of the structure, otherwise the function will fail with an error indicating that the parameter is invalid. All other members should be initialized to a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

See Also

[HttpGetServerTransferInfo](#)

INITDATA Structure

This structure contains information about the SocketTools library when the initialization function returns.

```
typedef struct _INITDATA
{
    DWORD        dwSize;
    DWORD        dwVersionMajor;
    DWORD        dwVersionMinor;
    DWORD        dwVersionBuild;
    DWORD        dwOptions;
    DWORD_PTR    dwReserved1;
    DWORD_PTR    dwReserved2;
    TCHAR        szDescription[128];
} INITDATA, *LPINITDATA;
```

Members

dwSize

Size of this structure. This structure member must be set to the size of the structure prior to calling the initialization function. Failure to do so will cause the function to fail.

dwVersionMajor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the major version number for the library.

dwVersionMinor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the minor version number for the library.

dwVersionBuild

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the build number for the library.

dwOptions

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved1

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved2

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

szDescription

A null terminated string which describes the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

SECURITYCREDENTIALS Structure

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on

	what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that

will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A pointer to a null terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A pointer to a null terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A pointer to a null terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A pointer to a null terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
------------	-------------

CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
----	---

MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
----	--

ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.
------	--

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

A pointer to a null terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used for SSH connections

and should always be NULL when establishing a secure connection using SSL or TLS.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* member should be set to CREDENTIAL_STORE_FILENAME and the *lpzCertStore* member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS#12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the *lpzPassword* member or the library will be unable to access the certificate.

Note that the *lpzUserName* and *lpzPassword* members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

The TLS 1.1 and TLS 1.2 protocols are only supported on Windows 7, Windows Server 2008 R2 and later versions of the platform. If these options are specified and the application is running on Windows XP or Windows Vista, the protocol version will be downgraded to TLS 1.0 for backwards compatibility with those platforms.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

Internet Control Message Protocol Library

Determine if a remote host is reachable and how packets of data are routed to that system.

Reference

- [Functions](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

File Name	CSICMV10.DLL
Version	10.0.1468.2518
LibID	A8CDC210-0CEC-4A3B-A367-A649CEEA7419
Import Library	CSICMV10.LIB
Dependencies	None
Standards	RFC 792

Overview

The Internet Control Message Protocol (ICMP) is commonly used to determine if a remote host is reachable and how packets of data are routed to that system. Users are most familiar with this protocol as it is implemented in the ping and traceroute command line utilities. The ping command is used to check if a system is reachable and the amount of time that it takes for a packet of data to make a round trip from the local system, to the remote host and then back again. The traceroute command is used to trace the route that a packet of data takes from the local system to the remote host, and can be used to identify potential problems with overall throughput and latency. The library can be used to build in this type of functionality in your own applications, giving you the ability to send and receive ICMP echo datagrams in order to perform your own analysis.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location

on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Internet Control Message Protocol Functions

Function	Description
IcmpCancel	Cancel the current blocking operation
IcmpCloseHandle	Close the current client session
IcmpCreateHandle	Create a new handle for the current client session
IcmpDisableEvents	Disable asynchronous event notification
IcmpDisableTrace	Disable logging of socket-level calls to the trace log
IcmpEcho	Send one or more echo datagrams to the specified host
IcmpEnableEvents	Enable asynchronous event notification
IcmpEnableTrace	Enable logging of socket function calls to a file
IcmpEventProc	Callback function that processes events generated by the client
IcmpFormatAddress	Convert a numeric IP address to a string
IcmpFreezeEvents	Suspend or resume event handling by the calling process
IcmpGetErrorString	Return a description for the specified error code
IcmpGetHostAddress	Return the current host IP address
IcmpGetHostName	Return the current host name
IcmpGetLastError	Return the last error code
IcmpGetPacketSize	Return the ICMP datagram packet size
IcmpGetRecvCount	Return the number of packets received
IcmpGetSendCount	Return the number of packets sent
IcmpGetSequenceId	Return the current sequence identifier
IcmpGetTimeout	Return the number of milliseconds until an operation times out
IcmpGetTimeToLive	Return the current time-to-live for the ICMP datagram
IcmpGetTripTime	Return the current trip time statistics
IcmpInitialize	Initialize the library and validate the specified license key at runtime
IcmpIsBlocking	Determine if the client is blocked, waiting for information
IcmpRecvEcho	Read an ICMP datagram returned by the remote host
IcmpRegisterEvent	Register an event callback function
IcmpReset	Reset the current client state
IcmpResolveAddress	Resolve an IP address into a fully qualified host name
IcmpSendEcho	Send an ICMP datagram to the specified host
IcmpSetHostAddress	Set the IP address of the host to receive the next datagram
IcmpSetHostName	Set the name of the host to receive the next datagram

IcmpSetLastError	Set the last error code
IcmpSetPacketSize	Set the ICMP datagram packet size
IcmpSetSequenceId	Set the sequence identifier for the next datagram
IcmpSetTimeout	Set the number of milliseconds until an operation times out
IcmpSetTimeToLive	Set the time-to-live for the next datagram
IcmpTraceRoute	Trace the route from the local system to a remote host
IcmpUninitialize	Terminate use of the library by the application

IcmpCancel Function

```
INT WINAPI IcmpCancel(  
    HCLIENT hClient  
);
```

The **IcmpCancel** function cancels any outstanding blocking operation in the client, causing the blocking function to fail. The application may then retry the operation or terminate the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

When the **IcmpCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: csicmv10.lib

See Also

[IcmpIsBlocking](#)

IcmpCloseHandle Function

```
INT WINAPI IcmpCloseHandle(  
    HCLIENT hClient  
);
```

The **IcmpCloseHandle** function closes the socket and releases the memory allocated for the client session.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IcmpCreateHandle](#), [IcmpInitialize](#), [IcmpUninitialize](#)

IcmpCreateHandle Function

```
HCLIENT WINAPI IcmpCreateHandle(  
    UINT nPacketSize,  
    UINT nTimeToLive,  
    DWORD dwTimeout,  
    DWORD dwReserved,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **IcmpCreateHandle** function creates a client handle for sending and receiving ICMP echo datagrams. If an event notification window is specified, the client will be notified when a network event occurs.

Parameters

nPacketSize

An unsigned integer which specifies the default packet size used when generating ICMP echo datagrams. The minimum packet size is 32 bytes and the maximum size is 65,535 bytes.

nTimeToLive

An unsigned integer which specifies the default time-to-live for ICMP echo datagrams. This determines the maximum number of times that a packet will be routed from one system to another while enroute to its destination. The minimum time-to-live value is 1, the maximum is 255. The recommend value for this parameter is 255, and typical applications should use a time-to-live value of at least 30.

dwTimeout

An unsigned integer which specifies the maximum number of milliseconds to wait before the current operation times out.

dwReserved

A reserved parameter. This value should always be zero.

hEventWnd

The handle to an asynchronous notification window. This window receives messages which notify the client when asynchronous network events occur. If asynchronous event notification is not required, this parameter may be NULL.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the *hEventWnd* parameter is NULL, this parameter should be specified as WM_NULL.

Return Value

If the function succeeds, the return value is a handle to the client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **IcmpGetLastError**.

Remarks

The **IcmpCreateHandle** function creates a client handle that is used with subsequent calls to the library. This library uses a special type of socket called a raw socket, which is created to send and receive ICMP echo datagrams. Raw socket support is optional under the Windows Sockets specification, and may not be available if a non-standard networking libraries are used or may

only be available to privileged accounts.

If the *hEventWnd* parameter is not NULL, the client operates in asynchronous mode and messages will be posted to the notification window when a network event occurs. When a message is posted to the window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
ICMP_EVENT_ECHO	The client has generated an ICMP echo datagram and has sent it to the specified host. If the datagram is received, the remote host should generate a reply and return it to the sender.
ICMP_EVENT_REPLY	The client has received an ICMP echo reply datagram from the remote host. At this point the client can collect statistical information.
ICMP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation.
ICMP_EVENT_CANCEL	The current operation has been canceled. The client application may attempt to retry the operation or close the handle.

To cancel asynchronous notification and return the client to a blocking mode, use the **IcmpDisableEvents** function.

The ability to create and send ICMP echo datagrams is limited to privileged users. Non-administrator users will receive an error if they attempt to create a client handle. On Windows NT it is possible to disable this security check by creating or modifying the system registry. Microsoft Knowledge Base article 195445 has additional information and instructions for making this change.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: csicmv10.lib

See Also

[IcmpCloseHandle](#), [IcmpInitialize](#), [IcmpUninitialize](#)

IcmpDisableEvents Function

```
INT WINAPI IcmpDisableEvents(  
    HCLIENT hClient  
);
```

The **IcmpDisableEvents** function disables all event notification, including event callbacks. Any pending events are deleted.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

This function affects both event notification and event callbacks. Any outstanding events in the message queue should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: csicmv10.lib

See Also

[IcmpEnableEvents](#), [IcmpFreezeEvents](#), [IcmpRegisterEvent](#)

IcmpDisableTrace Function

```
BOOL WINAPI IcmpDisableTrace();
```

The **IcmpDisableTrace** function disables the logging of socket function calls to the trace log file.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[IcmpEnableTrace](#)

IcmpEcho Function

```
INT WINAPI IcmpEcho(  
    LPCTSTR lpszHostName,  
    UINT nIterations,  
    UINT nPacketSize,  
    DWORD dwTimeout,  
    DWORD dwReserved,  
    LPICMPTIME lpTime  
);
```

The **IcmpEcho** function sends one or more ICMP echo datagrams, collecting information about the reliability and latency of a connection between the local and remote host.

Parameters

lpszHostName

A pointer to a string which specifies the fully qualified domain name of the remote host, or the IP address in dotted notation.

nIterations

An unsigned integer value which specifies the number of echo datagrams that will be sent to the remote host. The minimum value for this parameter is 1 and the maximum value is 512.

nPacketSize

An unsigned integer value which specifies the size of the echo datagram in bytes. The minimum size is 1 byte and the maximum size is 65,535 bytes. It is recommended that most applications use the minimum size of 32 bytes for this parameter.

dwTimeout

An unsigned integer which specifies the number of milliseconds the function will wait for a response to an echo datagram.

dwReserved

A reserved parameter. This value should always be zero.

lpTime

A pointer to an [ICMPTIME](#) structure which will contain the minimum, maximum and average round trip times for the echo datagrams sent and received.

Return Value

If the function succeeds, the return value is the number of replies received from the remote host. If the function fails, the return value is `ICMP_ERROR`. To get extended error information, call **IcmpGetLastError**.

Remarks

The **IcmpEcho** function sends a series of ICMP echo datagrams to the specified host, providing a simplified interface for pinging a remote system. If the function returns the same value as the number of iterations, then replies were received for all of the echo datagrams that were sent. This would typically indicate that the client can establish a reliable connection to the server; the values returned in the `ICMPTIME` structure provide information on the latency between the two hosts. Higher average time values would indicate greater latency and reduced throughput between the systems. If the function returns a value less than the specified number of iterations, this indicates that replies were not received for one or more of the echo datagrams. This may indicate that there

are problems routing data between the local and remote host. A return value of zero indicates that there was no response to the echo datagrams. The remote host may not exist or may not be available.

The failure for a host to respond to an ICMP echo datagram may not indicate a problem with the remote system. In some cases, a router between the local and remote host may be malfunctioning or discarding the datagrams. Systems can also be configured to specifically ignore ICMP echo datagrams and not respond to them; this is often a security measure to prevent certain kinds of Denial of Service attacks.

The ability to send ICMP datagrams may be restricted to users with administrative privileges, depending on the policies and configuration of the local system. If you are unable to send or receive any ICMP datagrams, it is recommended that you check the firewall settings and any third-party security software that could impact the normal operation of this component.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[IcmpCreateHandle](#), [IcmpGetTripTime](#), [IcmpRecvEcho](#), [IcmpSendEcho](#), [IcmpTraceRoute](#)

IcmpEnableEvents Function

```
INT WINAPI IcmpEnableEvents(  
    HCLIENT hClient,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **IcmpEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and is retained for backwards compatibility. Applications should use the **IcmpRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the event notification window. This window receives a user-defined message which specifies the event that has occurred. If this value is NULL, event notification is disabled.

uEventMsg

An unsigned integer which specifies the user-defined message that is sent when an event occurs. This parameter's value must be greater than the value of WM_USER.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
ICMP_EVENT_ECHO	The client has generated an ICMP echo datagram and has sent it to the specified host. If the datagram is received, the remote host should generate a reply and return it to the sender.
ICMP_EVENT_REPLY	The client has received an ICMP echo reply datagram from the remote host. At this point the client can collect statistical information.
ICMP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation.
ICMP_EVENT_CANCEL	The current operation has been canceled. The client application may attempt to retry the operation or close the handle.

To cancel asynchronous notification and return the client to a blocking mode, use the **IcmpDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csicmv10.lib

See Also

[IcmpDisableEvents](#), [IcmpFreezeEvents](#), [IcmpRegisterEvent](#)

IcmpEnableTrace Function

```
BOOL WINAPI IcmpEnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **IcmpEnableTrace** function enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the remote host.

Trace function logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IcmpDisableTrace](#)

IcmpEventProc Function

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

The **IcmpEventProc** function is an application-defined callback function that processes events generated by the calling process.

Parameters

hClient

The handle to the client session.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **IcmpRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **IcmpRegisterEvent** function. The **IcmpEventProc** function is a placeholder for the application-defined function name.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[IcmpDisableEvents](#), [IcmpEnableEvents](#), [IcmpFreezeEvents](#), [IcmpRegisterEvent](#)

IcmpFormatAddress Function

```
INT WINAPI IcmpFormatAddress(  
    DWORD dwAddress,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

The **IcmpFormatAddress** function converts a numeric IP address to a string.

Parameters

dwAddress

A unsigned 32-bit integer which specifies the IP address to be converted into a string.

lpszAddress

A pointer to a null-terminated array of characters which will contain the converted IP address in dot-notation. This string should be at least 16 characters in length.

nMaxLength

The maximum number of characters which may be copied in to the string buffer.

Return Value

If the function succeeds, the return value is the length of the string buffer. If the function fails, the return value is zero. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IcmpGetHostAddress](#), [IcmpGetHostName](#), [IcmpResolveAddress](#)

IcmpFreezeEvents Function

```
INT WINAPI IcmpFreezeEvents(  
    HCLIENT hClient,  
    BOOL bFreeze  
);
```

The **IcmpFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[IcmpDisableEvents](#), [IcmpEnableEvents](#), [IcmpRegisterEvent](#)

IcmpGetErrorString Function

```
INT WINAPI IcmpGetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);
```

The **IcmpGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **IcmpGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is 0, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csicmv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IcmpGetLastError](#), [IcmpSetLastError](#)

IcmpGetHostAddress Function

```
INT WINAPI IcmpGetHostAddress(  
    HCLIENT hClient,  
    LPTSTR lpszHostAddress,  
    INT cbHostAddress  
);
```

The **IcmpGetHostAddress** copies the IP address of the current host into the specified buffer as a string using dot-notation.

Parameters

hClient

A handle to the client session.

lpszHostAddress

A pointer to the buffer that will contain the IP address of the current remote host in dot-notation. This buffer should be at least 16 characters in length.

cbHostAddress

The maximum number of characters that may be copied into the buffer, including the terminating null character character.

Return Value

If the function succeeds, the return value is the length of the address string. If the return value is zero, this indicates that no remote host has been specified. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IcmpFormatAddress](#), [IcmpGetHostName](#), [IcmpResolveAddress](#), [IcmpSetHostAddress](#),
[IcmpSetHostName](#)

IcmpGetHostName Function

```
INT WINAPI IcmpGetHostName(  
    HCLIENT hClient,  
    LPTSTR lpszHostName,  
    INT cbHostName  
);
```

The **IcmpGetHostName** copies the name of the current host into the specified buffer.

Parameters

hClient

A handle to the client session.

lpszHostName

A pointer to the buffer that will contain the name of the current remote host. This buffer should be at least 64 characters in length.

cbHostName

The maximum number of characters that may be copied into the buffer, including the terminating null character character.

Return Value

If the function succeeds, the return value is the length of the host name string. If the return value is zero, this indicates that no remote host has been specified. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IcmpFormatAddress](#), [IcmpGetHostAddress](#), [IcmpResolveAddress](#), [IcmpSetHostAddress](#), [IcmpSetHostName](#)

IcmpGetLastError Function

```
DWORD WINAPI IcmpGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **IcmpSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **IcmpGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **IcmpSetLastError(0)** when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or ICMP_ERROR. Those functions which call **IcmpSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[IcmpGetErrorString](#), [IcmpSetLastError](#)

IcmpGetPacketSize Function

```
UINT WINAPI IcmpGetPacketSize(  
    HCLIENT hClient  
);
```

The **IcmpGetPacketSize** function returns the size of the ICMP echo datagram that will be sent to the remote host. The minimum packet size is 1 byte, the maximum packet size is 65,535 bytes.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is the size of the datagram. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IcmpGetRecvCount](#), [IcmpGetSendCount](#), [IcmpGetSequenceId](#), [IcmpGetTimeToLive](#),
[IcmpGetTripTime](#), [IcmpSetPacketSize](#), [IcmpSetSequenceId](#), [IcmpSetTimeToLive](#)

IcmpGetRecvCount Function

```
INT WINAPI IcmpGetRecvCount(  
    HCLIENT hClient  
);
```

The **IcmpGetRecvCount** function returns the number of replies sent to the client from the remote host.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is the number of replies. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IcmpGetPacketSize](#), [IcmpGetSendCount](#), [IcmpGetSequenceId](#), [IcmpGetTimeToLive](#), [IcmpGetTripTime](#), [IcmpSetSequenceId](#), [IcmpSetTimeToLive](#)

IcmpGetSendCount Function

```
INT WINAPI IcmpGetSendCount(  
    HCLIENT hClient  
);
```

The **IcmpGetSendCount** function returns the number of datagrams sent to the remote host by the client.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is the number of datagrams sent by the client. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IcmpGetPacketSize](#), [IcmpGetRecvCount](#), [IcmpGetSequenceId](#), [IcmpGetTimeToLive](#), [IcmpGetTripTime](#), [IcmpSetSequenceId](#), [IcmpSetTimeToLive](#)

IcmpGetSequenceId Function

```
INT WINAPI IcmpGetSequenceId(  
    HCLIENT hClient  
);
```

The **IcmpGetSequenceId** function returns the sequence identifier for the last ICMP echo datagram received by the client.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is the sequence identifier. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IcmpGetPacketSize](#), [IcmpGetRecvCount](#), [IcmpGetSendCount](#), [IcmpGetTimeToLive](#), [IcmpGetTripTime](#), [IcmpSetSequenceId](#), [IcmpSetTimeToLive](#)

IcmpGetTimeout Function

```
DWORD WINAPI IcmpGetTimeout(  
    HCLIENT hClient  
);
```

The **IcmpGetTimeout** function returns the number of milliseconds the client will wait for a response from the remote host. Once the specified number of milliseconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in milliseconds. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[IcmpIsBlocking](#), [IcmpSetTimeout](#)

IcmpGetTimeToLive Function

```
INT WINAPI IcmpGetTimeToLive(  
    HCLIENT hClient  
);
```

The **IcmpGetTimeToLive** function returns the time-to-live for the last ICMP echo datagram received by the client.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is the time-to-live for the last datagram. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

The time-to-live (TTL) value is specified in the IP header of a datagram, and is used to control the number of routers that the datagram is passed through. Each router that handles the datagram decrements the TTL value by one. When it drops to zero, a datagram is returned to the sender, specifying that the TTL has been exceeded. The default value is 255.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[IcmpGetPacketSize](#), [IcmpGetRecvCount](#), [IcmpGetSendCount](#), [IcmpGetSequenceId](#), [IcmpGetTripTime](#), [IcmpSetSequenceId](#), [IcmpSetTimeToLive](#)

IcmpGetTripTime Function

```
INT WINAPI IcmpGetTripTime(  
    HCLIENT hClient,  
    LPICMP_TIME lpIcmpTime  
);
```

The **IcmpGetTripTime** returns round-trip statistics for the datagrams sent to the current remote host.

Parameters

hClient

A handle to the client session.

lpIcmpTime

A pointer to an [ICMP_TIME](#) data structure which will contain the round-trip statistics for the current remote host.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `ICMP_ERROR`. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csicmv10.lib`

See Also

[IcmpGetPacketSize](#), [IcmpGetRecvCount](#), [IcmpGetSendCount](#), [IcmpGetSequenceId](#), [IcmpGetTimeToLive](#), [IcmpSetSequenceId](#), [IcmpSetTimeToLive](#)

IcmpInitialize Function

```
BOOL WINAPI IcmpInitialize(  
    LPCTSTR lpszLicenseKey,  
    LPINITDATA lpData  
);
```

The **IcmpInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpData

Pointer to an INITDATA data structure. This parameter may be NULL if the initialization data for the library is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **IcmpGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must insure that this function is called with a valid license key.

If the *lpData* argument is specified, it must point to an INITDATA structure which has been initialized by setting the *dwSize* member to the size of the structure. All other structure members should be set to zero. If the function is successful, then the INITDATA structure will be filled with identifying information about the library.

Although it is only required that **IcmpInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **IcmpUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **IcmpInitialize** or **IcmpUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

IcmpIsBlocking Function

```
BOOL WINAPI IcmpIsBlocking(  
    HCLIENT hClient  
);
```

The **IcmpIsBlocking** function is used to determine if the client is currently performing a blocking operation.

Parameters

hClient

Handle to the client session.

Return Value

If the client is performing a blocking operation, the function returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the function returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking function may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **IcmpIsBlocking** function can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csicmv10.lib`

See Also

[IcmpCancel](#)

IcmpRecvEcho Function

```
INT WINAPI IcmpRecvEcho(  
    HCLIENT hClient  
);
```

The **IcmpRecvEcho** function reads the reply to an ICMP echo datagram generated by the client. This function should only be called by asynchronous client sessions in response to an event notification that a datagram has been received.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[IcmpGetRecvCount](#), [IcmpGetSequenceId](#), [IcmpGetTimeToLive](#), [IcmpSendEcho](#)

IcmpRegisterEvent Function

```
INT WINAPI IcmpRegisterEvent(  
    HCLIENT hClient,  
    UINT nEventId,  
    INETEVENTPROC LpfnEvent,  
    DWORD_PTR dwParam  
);
```

The **IcmpRegisterEvent** function registers a callback function for the specified event.

Parameters

hClient

Handle to client session.

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One or more of the following values may be used:

Constant	Description
ICMP_EVENT_ECHO	The client has generated an ICMP echo datagram and has sent it to the specified host. If the datagram is received, the remote host should generate a reply and return it to the sender.
ICMP_EVENT_REPLY	The client has received an ICMP echo reply datagram from the remote host. At this point the client can collect statistical information.
ICMP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation.
ICMP_EVENT_CANCEL	The current operation has been canceled. The client application may attempt to retry the operation or close the handle.

lpfnEvent

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **IcmpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

The **IcmpRegisterEvent** function associates a callback function with a specific event. The event

handler is an **IcmpEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[IcmpDisableEvents](#), [IcmpEnableEvents](#), [IcmpEventProc](#), [IcmpFreezeEvents](#)

IcmpReset Function

```
INT WINAPI IcmpReset(  
    HCLIENT hClient  
);
```

The **IcmpReset** function resets the client session, clearing the packet trip statistics, time-to-live, sequence identifier, send and receive counts.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[IcmpSetHostAddress](#), [IcmpSetHostName](#), [IcmpSetSequenceId](#), [IcmpSetTimeToLive](#)

IcmpResolveAddress Function

```
INT WINAPI IcmpResolveAddress(  
    DWORD dwAddress,  
    LPTSTR lpszHostName,  
    INT nMaxLength  
);
```

The **IcmpResolveAddress** function resolves a numeric IP address into a fully qualified domain name.

Parameters

dwAddress

The IP address to be resolved, specified as an unsigned 32-bit integer in network byte order.

lpszHostName

A pointer to a buffer that will contain a null-terminated string that specifies the fully qualified domain name for the host. It is recommended that this buffer be at least 64 characters in length.

nMaxLength

The maximum number of characters that can be copied into the string buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the host name. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

The **IcmpResolveAddress** function is used to perform a reverse DNS lookup, converting a numeric IP address into a fully qualified domain name. This is useful for functions like **IcmpTraceRoute**, which return host addresses as numeric values in network byte order. If the reverse DNS lookup fails because there is no PTR record for the given IP address, a printable form of the address in dotted notation will be returned in the string buffer.

Calling this function will cause the thread to block until the IP address is resolved, or until the DNS query times out because there is no reverse record for the address. In some cases, a reverse lookup can take several seconds.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csicmv10.lib

See Also

[IcmpGetHostAddress](#), [IcmpGetHostName](#), [IcmpSetHostAddress](#), [IcmpSetHostName](#), [IcmpTraceRoute](#)

IcmpSendEcho Function

```
INT WINAPI IcmpSendEcho(  
    HCLIENT hClient  
);
```

The **IcmpSendEcho** function sends an ICMP echo datagram to the remote host.

Parameters

hClient

A handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

For asynchronous client sessions, this function returns immediately. Otherwise, the client enters a blocking state and waits for a reply from the remote host. The function returns when a reply has been received, or the operation times-out.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IcmpEcho](#), [IcmpGetSendCount](#), [IcmpRecvEcho](#), [IcmpSetSequenceId](#), [IcmpSetTimeToLive](#), [IcmpTraceRoute](#)

IcmpSetHostAddress Function

```
INT WINAPI IcmpSetHostAddress(  
    HCLIENT hClient,  
    DWORD dwAddress  
);
```

The **IcmpSetHostAddress** function specifies the IP address of the host to receive an ICMP echo datagram. If the address specifies a new host, the current client statistics are reset.

Parameters

hClient

A handle to the client session.

dwAddress

The IP address of the remote host as a 32-bit integer value, specified in network byte order.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

To specify a remote host name or an IP address as a string in dot notation, use the **IcmpSetHostName** function instead.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[IcmpGetHostAddress](#), [IcmpGetHostName](#), [IcmpSetHostName](#)

IcmpSetHostName Function

```
INT WINAPI IcmpSetHostName(  
    HCLIENT hClient,  
    LPCTSTR lpszHostName  
);
```

The **IcmpSetHostName** function specifies the name of the host to receive an ICMP echo datagram. If the name specifies a new host, the current client statistics are reset.

Parameters

hClient

A handle to the client session.

lpszHostName

A pointer to a string which specifies the name, or IP address in dot-notation, of the remote host.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[IcmpGetHostAddress](#), [IcmpGetHostName](#), [IcmpSetHostAddress](#)

IcmpSetLastError Function

```
VOID WINAPI IcmpSetLastError(  
    DWORD dwErrorCode  
);
```

The **IcmpSetLastError** function sets the error code for the current thread. This function is typically used to clear the last error by passing a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or ICMP_ERROR. Those functions which call **IcmpSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **IcmpGetLastError** function. The use of **IcmpGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[IcmpGetErrorString](#), [IcmpGetLastError](#)

IcmpSetPacketSize Function

```
UINT WINAPI IcmpSetPacketSize(  
    HCLIENT hClient,  
    UINT nPacketSize  
);
```

The **IcmpSetPacketSize** function sets the size of the ICMP datagram packet that is sent to the remote host.

Parameters

hClient

Handle to the client session.

nPacketSize

Size of the ICMP datagram packet in bytes. The minimum packet size is 1 byte and the maximum packet size is 65,535 bytes.

Return Value

If the function succeeds, the return value is the previous ICMP datagram packet size. If the function fails, the return value is zero. To get extended error information, call **IcmpGetLastError**.

Remarks

Note that packet sizes over 512 bytes may not be supported by your networking software or configuration. It is recommended that most applications use the minimum packet size of 32 bytes.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[IcmpGetPacketSize](#), [IcmpSetSequenceId](#), [IcmpSetTimeToLive](#)

IcmpSetSequenceId Function

```
INT WINAPI IcmpSetSequenceId(  
    HCLIENT hClient,  
    INT nSequenceId  
);
```

The **IcmpSetSequenceId** function sets the sequence identifier for the next ICMP echo datagram sent by the client. The default sequence identifier for the first datagram is one.

Parameters

hClient

A handle to the client session.

nSequenceId

The sequence identifier for the next datagram sent by the client.

Return Value

If the function succeeds, the return value is the previous sequence identifier. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[IcmpGetPacketSize](#), [IcmpGetRecvCount](#), [IcmpGetSendCount](#), [IcmpGetSequenceId](#), [IcmpGetTimeToLive](#), [IcmpGetTripTime](#), [IcmpSetPacketSize](#), [IcmpSetTimeToLive](#)

IcmpSetTimeout Function

```
DWORD WINAPI IcmpSetTimeout(  
    HCLIENT hClient,  
    DWORD dwTimeout  
);
```

The **IcmpSetTimeout** function sets the number of milliseconds the client will wait for a response from the remote host. Once the specified number of milliseconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

dwTimeout

The number of milliseconds until a blocking operation fails.

Return Value

If the function succeeds, the return value is the value of the timeout before the function was called.

If the function fails, the return value is ICMP_ERROR. To get extended error information, call

IcmpGetLastError.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[IcmpCreateHandle](#), [IcmpGetTimeout](#)

IcmpSetTimeToLive Function

```
INT WINAPI IcmpSetTimeToLive(  
    HCLIENT hClient,  
    INT nTimeToLive  
);
```

The **IcmpSetTimeToLive** function sets the maximum time-to-live for the next ICMP datagram sent by the client.

Parameters

hClient

A handle to the client session.

nTimeToLive

The time-to-live value for the next ICMP echo datagram.

Return Value

If the function succeeds, the return value is the previous time-to-live value. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

The time-to-live (TTL) value is specified in the IP header of a datagram, and is used to control the number of routers that the datagram is passed through. Each router that handles the datagram decrements the TTL value by one. When it drops to zero, a datagram is returned to the sender, specifying that the TTL has been exceeded.

Calling this function changes the default TTL value for all subsequent ICMP datagrams sent by the library, with the default value being 255. Note that not all Windows Sockets implementations support setting the time-to-live value.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[IcmpGetPacketSize](#), [IcmpGetRecvCount](#), [IcmpGetSendCount](#), [IcmpGetSequenceId](#), [IcmpGetTimeToLive](#), [IcmpGetTripTime](#), [IcmpSetPacketSize](#), [IcmpSetSequenceId](#)

IcmpTraceRoute Function

```
INT WINAPI IcmpTraceRoute(  
    LPCTSTR lpszHostName,  
    UINT nMaxHops,  
    DWORD dwTimeout,  
    DWORD dwReserved,  
    LPICMPTRACE lpTrace,  
    LPDWORD lpdwSize  
);
```

The **IcmpTraceRoute** function sends a series of ICMP echo datagrams to trace the route taken from the local system to the remote host.

Parameters

lpszHostName

A pointer to a string which specifies the fully qualified domain name of the remote host, or the IP address in dotted notation.

nMaxHops

An unsigned integer which specifies the maximum number of routers the datagram will be forwarded through (the number of hops) to the remote host. The minimum value is 1 and the maximum value is 255. It is recommended that most applications specify a value of at least 30.

dwTimeout

An unsigned integer which specifies the number of milliseconds the function will wait for a response to an echo datagram.

dwReserved

A reserved parameter. This value should always be zero.

lpTrace

A pointer to an array of [ICMPTRACE](#) structures which will contain information about each intermediate host between the local and remote system. The number of structures must be at least the same as the maximum number of hops specified by the *nMaxHops* parameter. To determine the total number of bytes that must be allocated for the structure array, this parameter can be passed as NULL and the total size will be returned in the *lpdwSize* parameter.

lpdwSize

A pointer to an unsigned integer which should be initialized to the size of the ICMPTRACE array passed to the function. If the *lpTrace* parameter is NULL, the function will calculate the size of the array buffer that must be allocated and return that value. If both the *lpTrace* and *lpdwSize* parameters are NULL, no statistical information about the intermediate hosts will be collected.

Return Value

If the function succeeds, the return value is the number of intermediate hosts between the local and remote system. If the function fails, the return value is ICMP_ERROR. To get extended error information, call **IcmpGetLastError**.

Remarks

The **IcmpTraceRoute** function sends a series of ICMP echo datagrams to the specified host, adjusting the time-to-live value to determine the intermediate hosts that route the packet. The function returns the number of hops to the remote host.

It is important to note that the failure of an intermediate host to respond to an ICMP echo datagram may not indicate a problem with the remote system. Systems can be configured to specifically ignore ICMP echo datagrams and not respond to them; this is often a security measure to prevent certain kinds of Denial of Service attacks.

The ability to send ICMP datagrams may be restricted to users with administrative privileges, depending on the policies and configuration of the local system. If you are unable to send or receive any ICMP datagrams, it is recommended that you check the firewall settings and any third-party security software that could impact the normal operation of this component.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csicmv10.lib

See Also

[IcmpCreateHandle](#), [IcmpEcho](#), [IcmpRecvEcho](#), [IcmpResolveAddress](#), [IcmpSendEcho](#),
[IcmpSetTimeToLive](#)

IcmpUninitialize Function

```
VOID WINAPI IcmpUninitialize();
```

The **IcmpUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **IcmpInitialize** call before it can call any of the other the library functions. When it has completed the use of library, the application must call **IcmpUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There must be a call to **IcmpUninitialize** for every successful call to **IcmpInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csicmv10.lib

See Also

[IcmpCloseHandle](#), [IcmpCreateHandle](#), [IcmpInitialize](#)

Internet Control Message Protocol Data Structures

- ICMPTIME
- ICMPTRACE
- INITDATA
- INTERNET_ADDRESS
- SYSTEMTIME

ICMPTIME Structure

This structure is used by the [IcmpGetTripTime](#) function to return the round-trip times of an ICMP datagram.

```
typedef struct _ICMPTIME {
    DWORD dwTripAverage;
    DWORD dwTripMaximum;
    DWORD dwTripMinimum;
    DWORD dwTripTime;
} ICMPTIME, *LPICMPTIME;
```

Members

dwTripAverage

The average round-trip time in milliseconds for all datagrams sent to the current host.

dwTripMaximum

The maximum round-trip time in milliseconds for all datagrams sent to current host.

dwTripMinimum

The minimum round-trip time in milliseconds for all datagrams sent to the current host.

dwTripTime

The current round-trip time in milliseconds for the last datagram sent to the current host.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

ICMPTRACE Structure

This structure is used by the [IcmpTraceRoute](#) function to return the route of an ICMP datagram.

```
typedef struct _ICMPTRACE
{
    UINT          nDistance;
    DWORD        dwHostAddress;
    DWORD        dwTripAverage;
    DWORD        dwTripMaximum;
    DWORD        dwTripMinimum;
} ICMPTRACE, *LPICMPTRACE;
```

Members

nDistance

The distance from the local host to the remote host for this route.

dwHostAddress

An unsigned integer which specifies the IP address of the remote host in network byte order.

dwTripAverage

The average round-trip time in milliseconds for all datagrams sent to the specified host.

dwTripMaximum

The maximum round-trip time in milliseconds for all datagrams sent to specified host.

dwTripMinimum

The minimum round-trip time in milliseconds for all datagrams sent to the specified host.

dwTripTime

The current round-trip time in milliseconds for the last datagram sent to the specified host.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

INITDATA Structure

This structure contains information about the SocketTools library when the initialization function returns.

```
typedef struct _INITDATA
{
    DWORD      dwSize;
    DWORD      dwVersionMajor;
    DWORD      dwVersionMinor;
    DWORD      dwVersionBuild;
    DWORD      dwOptions;
    DWORD_PTR  dwReserved1;
    DWORD_PTR  dwReserved2;
    TCHAR      szDescription[128];
} INITDATA, *LPINITDATA;
```

Members

dwSize

Size of this structure. This structure member must be set to the size of the structure prior to calling the initialization function. Failure to do so will cause the function to fail.

dwVersionMajor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the major version number for the library.

dwVersionMinor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the minor version number for the library.

dwVersionBuild

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the build number for the library.

dwOptions

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved1

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved2

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

szDescription

A null terminated string which describes the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

INTERNET_ADDRESS Structure

This structure represents a numeric IPv4 or IPv6 address in network byte order.

```
typedef struct _INTERNET_ADDRESS
{
    INT    ipFamily;
    BYTE  ipNumber[16];
} INTERNET_ADDRESS, *LPINTERNET_ADDRESS;
```

Members

ipFamily

An integer which identifies the type of IP address. It will be one of the following values:

Constant	Description
INET_ADDRESS_UNKNOWN	The address has not been specified or the bytes in the <i>ipNumber</i> array does not represent a valid address. Functions which populate this structure will use this value to indicate that the address cannot be determined.
INET_ADDRESS_IPV4	Specifies that the address is in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address is in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

ipNumber

A byte array which contains the numeric form of the IP address. This array is large enough to store both IPv4 (32 bit) and IPv6 (128 bit) addresses. The values are stored in network byte order.

Remarks

The **INTERNET_ADDRESS** structure is used by some functions to represent an Internet address in a binary format that is compatible with both IPv4 and IPv6 addresses. Applications that use this structure should consider it to be opaque, and should not modify the contents of the structure directly.

For compatibility with legacy applications that expect an IP address to be 32 bits and stored in an unsigned integer, you can copy the first four bytes of the *ipNumber* array using the **CopyMemory** function or equivalent. Note that if this is done, your application should always check the *ipFamily* member first to make sure that it is actually an IPv4 address.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

SYSTEMTIME Structure

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include windows.h.

Internet Message Access Protocol Library

Manage email messages and mailboxes on a mail server.

Reference

- [Functions](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

File Name	CSMAPV10.DLL
Version	10.0.1468.2518
LibID	E1419168-2803-4EF6-A0BC-48A71D9EEFD7
Import Library	CSMAPV10.LIB
Dependencies	None
Standards	RFC 3501

Overview

The Internet Message Access Protocol (IMAP) is an application protocol which is used to access a user's email messages which are stored on a mail server. However, unlike the Post Office Protocol (POP) where messages are downloaded and processed on the local system, the messages on an IMAP server are retained on the server and processed remotely. This is ideal for users who need access to a centralized store of messages or have limited bandwidth. For example, traveling salesmen who have notebook computers or mobile users on a wireless network would be ideal candidates for using IMAP.

The SocketTools IMAP library implements the current standard for this protocol, and provides functions to retrieve messages, or just certain parts of a message, create and manage mailboxes, search for specific messages based on certain criteria and so on. The API is designed as a superset of the Post Office Protocol API, so developers who are used to working with the POP3 library will find the IMAP library very easy to integrate into an existing application.

This library supports secure connections using the standard SSL and TLS protocols.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Internet Message Access Protocol Functions

Function	Description
ImapAsyncConnect	Connect asynchronously to the specified server
ImapAttachThread	Attach the specified client handle to another thread
ImapCancel	Cancel the current blocking operation
ImapCheckMailbox	Check the current mailbox for new messages
ImapCloseMessage	Close the current message
ImapCommand	Send a command to the server
ImapConnect	Connect to the specified server
ImapCopyMessage	Copy a message from the current mailbox to another mailbox
ImapCreateMailbox	Create a new mailbox on the server
ImapCreateMessage	Create a new message in the specified mailbox
ImapCreateSecurityCredentials	Allocate a structure to establish client security credentials
ImapDeleteMailbox	Delete the specified mailbox from the server
ImapDeleteMessage	Delete the specified message from the mailbox
ImapDeleteSecurityCredentials	Delete the specified client security credentials
ImapDisableEvents	Disable the event notification mechanism
ImapDisableTrace	Disable logging of socket function calls to the trace log
ImapDisconnect	Disconnect from the current server
ImapEnableEvents	Enable the client event notification mechanism
ImapEnableTrace	Enable logging of socket function calls to a file
ImapEnumMessages	Enumerate the messages in the current mailbox
ImapEventProc	Callback function that processes events generated by the client
ImapExamineMailbox	Select the specified mailbox in read-only mode
ImapExpungeMailbox	Remove messages that have been marked for deletion from the current mailbox
ImapFreezeEvents	Suspend and resume event handling by the client
ImapGetCapability	Return a string which identifies the capabilities of the server
ImapGetCurrentMailbox	Return the name of the currently selected mailbox
ImapGetDeletedMessages	Return the messages that have been marked for deletion
ImapGetErrorString	Return a description for the specified error code
ImapGetFirstMailbox	Return the first mailbox according to specified criteria
ImapGetHeaderValue	Return the value of the specified header field
ImapGetHeaderValueEx	Return the value of a header field in the specified message part
ImapGetIdleThreadId	Return the ID of the thread created to monitor the client session
ImapGetLastError	Return the last error code
ImapGetMailboxDelimiter	Get the path delimiter for the specified mailbox hierarchy

ImapGetMailboxSize	Return the size of the specified mailbox
ImapGetMailboxStatus	Return the status of the specified mailbox
ImapGetMailboxUID	Return the unique identifier for the specified mailbox
ImapGetMessage	Retrieve the specified message from the server
ImapGetMessageCount	Return the number of messages available in the mailbox
ImapGetMessageCountEx	Return the number of messages available in the mailbox
ImapGetMessageFlags	Return the status flags for the specified message
ImapGetMessageHeaders	Retrieve the specified message header from the server
ImapGetMessageHeadersEx	Retrieve the specified message header, UID and flags from the server
ImapGetMessageId	Return the message ID string for the specified message
ImapGetMessageParts	Return the number of MIME message parts in the specified message
ImapGetMessageSender	Return the address of the message sender
ImapGetMessageSize	Return the size of the specified message
ImapGetMessageUid	Return the unique identifier for the specified message
ImapGetNewMessages	Return a list of the new messages in the current mailbox
ImapGetNextMailbox	Return the next mailbox name on the server
ImapGetResultCode	Return the result code from the previous command
ImapGetResultString	Return the result string from the previous command
ImapGetSecurityInformation	Return security information about the current client connection
ImapGetStatus	Return the current status of the client
ImapGetTimeout	Return the number of seconds until an operation times out
ImapGetTransferStatus	Return data transfer statistics
ImapGetUnseenMessages	Return a list of messages from the current mailbox that have not been read
ImapIdle	Enables mailbox status monitoring for the client session
ImapIdleProc	Callback function that receives update notifications from the server
ImapInitialize	Initialize the library and validate the specified license key at runtime
ImapIsBlocking	Determine if the client is blocked, waiting for information
ImapIsConnected	Determine if the client is connected to the server
ImapIsReadable	Determine if data can be read from the server
ImapIsWritable	Determine if data can be written to the server
ImapLogin	Login to the server
ImapOpenMessage	Open the specified message for reading on the server
ImapOpenMessageEx	Open the specified message for reading on the server, with additional options
ImapRead	Read data returned by the server
ImapRegisterEvent	Register an event handler for the specified event
ImapRenameMailbox	Rename the specified mailbox
ImapReselectMailbox	Reselect the current mailbox and return updated status information

ImapSearchMailbox	Search the mailbox according to specified criteria
ImapSelectMailbox	Select the specified mailbox in read-write mode
ImapSetLastError	Set the last error code for the current thread
ImapSetMessageFlags	Set one or more status flags for the specified message
ImapSetTimeout	Set the number of seconds until an operation times out
ImapStoreMessage	Store the contents of a message to the specified file
ImapSubscribeMailbox	Subscribe to the specified mailbox
ImapUndeleteMessage	Undelete the specified message from the current mailbox
ImapUninitialize	Terminate use of the library by the application
ImapUnselectMailbox	Unselect the current mailbox and expunge any deleted messages
ImapUnsubscribeMailbox	Unsubscribe from the specified mailbox
ImapWrite	Write data to the server

ImapAsyncConnect Function

```
HCLIENT WINAPI ImapAsyncConnect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSECURITYCREDENTIALS LpCredentials,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **ImapAsyncConnect** function is used to establish a connection with the server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **ImapConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

lpszHostName

A pointer to a null terminated string which specifies the host name or IP address of the IMAP server.

nRemotePort

The port number the client should use to establish the connection. A value of zero specifies that default port 143 should be used, which is the standard port number assigned to the IMAP service. If the secure port number is specified, an implicit SSL/TLS connection will be established by default.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation

dwOptions

An unsigned integer value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
IMAP_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
IMAP_OPTION_IDENTIFY	This option specifies the client should identify itself to the server. If enabled, the client will send the ID command to the server as defined in RFC 2971. This option has no effect if the server does not support the ID command.
IMAP_OPTION_TUNNEL	This option specifies that a tunneled TCP

	<p>connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.</p>
IMAP_OPTION_TRUSTEDSITE	<p>This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.</p>
IMAP_OPTION_SECURE	<p>This option specifies that a secure connection should be established with the server and requires that the server support either the SSL or TLS protocol. This option is the same as specifying IMAP_OPTION_SECURE_EXPLICIT, which initiates the secure session using the STARTTLS command.</p>
IMAP_OPTION_SECURE_EXPLICIT	<p>This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using either the SSL or TLS protocol and the STARTTLS command.</p>
IMAP_OPTION_SECURE_IMPLICIT	<p>This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using either the SSL or TLS protocol, and the secure session must be negotiated immediately after the connection has been established.</p>
IMAP_OPTION_SECURE_FALLBACK	<p>This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.</p>
HTTP_OPTION_PREFER_IPV6	<p>This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.</p>
IMAP_OPTION_FREETHREAD	<p>This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access</p>

to the handle is synchronized across multiple threads.

lpCredentials

A pointer to a [SECURITYCREDENTIALS](#) structure which is used to establish the client credentials for a secure connection to the server. The function **ImapCreateSecurityCredentials** can be used to create this structure if necessary. If a standard non-secure connection is being established, or client credentials are not required by the server, this parameter can be NULL.

hEventWnd

The handle to the asynchronous notification window. This window receives messages which notify the client of various asynchronous client events that occur. Specifying this parameter and a message identifier causes the connection to be non-blocking. If this parameter is NULL, then a blocking connection is established.

uEventMsg

The message identifier that is used when an asynchronous client event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the *hEventWnd* parameter is NULL, this parameter should be specified as WM_NULL.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **ImapGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
IMAP_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
IMAP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
IMAP_EVENT_READ	Data is available to read by the client. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the calling process is in asynchronous mode.
IMAP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
IMAP_EVENT_TIMEOUT	The client has timed out while waiting for a response from the server. Note that under some circumstances this event can be generated for a non-blocking connection, such as when the client is establishing a secure connection.

IMAP_EVENT_CANCEL	The client has canceled the current operation.
IMAP_EVENT_COMMAND	The client has processed a command that was sent to the server. The result code and result string can be used to determine if the response to the command. The high word of the IParam parameter should be checked, since this notification message will also be posed if the command cannot be executed.
IMAP_EVENT_PROGRESS	This event notification is sent periodically during lengthy blocking operations, such as retrieving a complete message from the server.

To cancel asynchronous notification and return the client to a blocking mode, use the **ImapDisableEvents** function.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **ImapAttachThread** function.

Specifying the IMAP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapConnect](#), [ImapDisableEvents](#), [ImapDisconnect](#), [ImapEnableEvents](#), [ImapInitialize](#), [ImapUninitialize](#)

ImapAttachThread Function

```
DWORD WINAPI ImapAttachThread(  
    HCLIENT hClient  
    DWORD dwThreadId  
);
```

The **ImapAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **ImapAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **ImapAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **ImapCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **ImapUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

See Also

[ImapCancel](#), [ImapAsyncConnect](#), [ImapConnect](#), [ImapDisconnect](#), [ImapUninitialize](#)

ImapCancel Function

```
INT WINAPI ImapCancel(  
    HCLIENT hClient  
);
```

The **ImapCancel** function cancels any outstanding blocking operation in the client, causing the blocking function to fail. The application may then retry the operation or terminate the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

When the **ImapCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

See Also

[ImapIsBlocking](#)

ImapCheckMailbox Function

```
INT WINAPI ImapCheckMailbox(  
    HCLIENT hClient,  
    UINT *LpnMessages,  
    UINT *LpnUnseen  
);
```

The **ImapCheckMailbox** function requests that the server create a checkpoint of the currently selected mailbox, and returns the current number of messages and unseen messages.

Parameters

hClient

Handle to the client session.

lpnMessages

A pointer to an unsigned integer value which will contain the number of messages in the currently selected mailbox when the function returns.

lpnUnseen

A pointer to an unsigned integer value which will contain the number of unseen messages in the currently selected mailbox.

Return Value

If the function succeeds, it returns zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

When the client requests a checkpoint, the server may perform implementation-dependent housekeeping for that mailbox, such as updating the mailbox on disk with the current state of the mailbox in memory. On some systems this command has no effect other than to update the client with the current number of messages in the mailbox.

This function actually sends two IMAP commands. The first is the CHECK command, followed by the NOOP command to poll for any new messages that have arrived. In addition to polling the server for new messages, this command can also be used to ensure the idle timer on the server does not expire and force a disconnect from the client.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[ImapCreateMailbox](#), [ImapGetFirstMailbox](#), [ImapGetMailboxStatus](#), [ImapGetNextMailbox](#), [ImapRenameMailbox](#)

ImapCloseMessage Function

```
INT WINAPI ImapCloseMessage(  
    HCLIENT hClient  
);
```

The **ImapCloseMessage** function closes the current message.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapCloseMessage** function closes the current message. If there is any remaining data left to be read from the message, it will be read and discarded.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmav10.lib

See Also

[ImapCreateMessage](#), [ImapOpenMessage](#), [ImapOpenMessageEx](#)

ImapCommand Function

```
INT WINAPI ImapCommand(  
    HCLIENT hClient,  
    LPCTSTR lpszCommand,  
    LPCTSTR lpszParameter  
);
```

The **ImapCommand** function sends a command to the server. This function is typically used for site-specific commands not directly supported by the API.

Parameters

hClient

Handle to the client session.

lpszCommand

The command which will be executed by the server.

lpszParameter

An optional command parameter. If the command requires more than one parameter, then they should be combined into a single string, with a space separating each parameter. If the command does not accept any parameters, this value may be NULL.

Return Value

If the function succeeds, it returns an IMAP result code. If the command was successful, it returns IMAP_RESULT_OK. A return value of IMAP_RESULT_CONTINUE indicates the command was accepted and the caller should proceed with the next command. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

A list of valid commands can be found in the technical specification for the protocol. The current version of the protocol which is supported by this library is version 4rev1 as defined in RFC 3501.

Use the **ImapGetResultCode** function to determine the result of the command that was sent to the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapGetResultCode](#), [ImapGetResultString](#)

ImapConnect Function

```
HCLIENT WINAPI ImapConnect(  
    LPCTSTR lpszHostName,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSECURITYCREDENTIALS LpCredentials  
);
```

The **ImapConnect** function is used to establish a connection with the server. It supports connection options beyond those provided beyond **ImapConnect**.

Parameters

lpszHostName

A pointer to a null terminated string which specifies the host name or IP address of the IMAP server.

nRemotePort

The port number the client should use to establish the connection. A value of zero specifies that default port 143 should be used, which is the standard port number assigned to the IMAP service. If the secure port number is specified, an implicit SSL/TLS connection will be established by default.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
IMAP_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
IMAP_OPTION_IDENTIFY	This option specifies the client should identify itself to the server. If enabled, the client will send the ID command to the server as defined in RFC 2971. This option has no effect if the server does not support the ID command.
IMAP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
IMAP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the

	connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
IMAP_OPTION_SECURE	This option specifies that a secure connection should be established with the server and requires that the server support either the SSL or TLS protocol. This option is the same as specifying IMAP_OPTION_SECURE_EXPLICIT, which initiates the secure session using the STARTTLS command.
IMAP_OPTION_SECURE_EXPLICIT	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.
IMAP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using either the SSL or TLS protocol, and the secure session must be negotiated immediately after the connection has been established.
IMAP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
IMAP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
IMAP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpCredentials

A pointer to a [SECURITYCREDENTIALS](#) structure which is used to establish the client credentials for a secure connection to the server. The function **ImapCreateSecurityCredentials** can be used to create this structure if necessary. If a standard non-secure connection is being established, or client credentials are not required by the server, this parameter can be NULL.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is `INVALID_CLIENT`. To get extended error information, call **ImapGetLastError**.

Remarks

To prevent this function from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **ImapConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **ImapAttachThread** function.

Specifying the `IMAP_OPTION_FREETHREAD` option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmapi10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapDisconnect](#), [ImapInitialize](#), [ImapLogin](#)

ImapCopyMessage Function

```
INT WINAPI ImapCopyMessage(  
    HCLIENT hClient,  
    UINT nMessageId,  
    LPCTSTR lpszMailbox  
);
```

The **ImapCopyMessage** function copies a message from the current mailbox to the specified mailbox. The message is appended to the mailbox, and the message flags and internal date are preserved.

Parameters

hClient

Handle to the client session.

nMessageId

The message identifier which specifies which message is to be copied to the mailbox. This value must be greater than zero and specify a valid message number.

lpszMailbox

A pointer to a string which specifies the name of the mailbox that the message will be copied to. The mailbox must already exist, and the client must have the appropriate access rights to modify the mailbox.

Return Value

If the function succeeds, it returns a value of zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

If the mailbox does not exist, the function will fail. To create a new mailbox, use the **ImapCreateMailbox** function. A message can be copied within the same mailbox, in which case the server may flag it as a new message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapCreateMailbox](#), [ImapGetResultCode](#), [ImapGetResultString](#)

ImapCreateMailbox Function

```
INT WINAPI ImapCreateMailbox(  
    HCLIENT hClient,  
    LPCTSTR lpszMailbox  
);
```

The **ImapCreateMailbox** function creates a new mailbox on the server.

Parameters

hClient

Handle to the client session.

lpszMailbox

A pointer to a string which specifies the new mailbox to be created.

Return Value

If the function succeeds, it returns a value of zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

If the mailbox name is suffixed with the server's hierarchy delimiter, this indicates to the server that the client intends to create mailbox names under the specified name in the hierarchy. If superior hierarchical names are specified in the mailbox name, then the server may automatically create them as needed. For example, if the mailbox name "Mail/Office/Projects" is specified and "Mail/Office" does not exist, it may be automatically created by the server.

The special mailbox name INBOX is reserved, and cannot be created. It is recommended that mailbox names only consist of printable ASCII characters, and the special characters "*" and "%" should be avoided.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapDeleteMailbox](#), [ImapGetFirstMailbox](#), [ImapGetNextMailbox](#), [ImapRenameMailbox](#)

ImapCreateMessage Function

```
INT WINAPI ImapCreateMessage(  
    HCLIENT hClient,  
    LPCTSTR lpszMailbox,  
    DWORD dwReserved,  
    LPVOID lpMessage,  
    DWORD dwMessageSize,  
    DWORD dwFlags  
);
```

The **ImapCreateMessage** function creates a message, appending it to the contents of the specified mailbox.

Parameters

hClient

Handle to the client session.

lpszMailbox

A pointer to a string which specifies the name of the mailbox that the message will be created in. The mailbox must already exist, and the client must have the appropriate access rights to modify the mailbox.

dwReserved

A reserved parameter. This value should always be zero.

lpMessage

A pointer to the buffer which contains the message to be created.

dwMessageSize

An unsigned integer value which specifies the size of the message in bytes. If this value is zero, then it is assumed that the *lpMessage* parameter points to a string.

dwFlags

An unsigned integer that specifies one or more message flags. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
IMAP_FLAG_NONE	No value.
IMAP_FLAG_ANSWERED	The message has been answered.
IMAP_FLAG_DRAFT	The message is not completed and is marked as a draft copy.
IMAP_FLAG_URGENT	The message is flagged for urgent or special attention.
IMAP_FLAG_SEEN	The message has been read.

Return Value

If the function succeeds, it returns zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

If the mailbox does not exist, the function will fail. To create a new mailbox, use the **ImapCreateMailbox** function. If the message is created in the mailbox that is currently selected, the server may flag it as recent.

This function is typically used by applications to store messages which have already been sent to a user. After a message has been delivered using the SMTP protocol, that same message may be created in a mailbox on the IMAP server so that the user has access to those messages.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmav10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapCopyMessage](#), [ImapDeleteMessage](#), [ImapGetMessageFlags](#), [ImapSetMessageFlags](#)

ImapCreateSecurityCredentials Function

```
BOOL WINAPI ImapCreateSecurityCredentials(  
    DWORD dwProtocol,  
    DWORD dwOptions,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName,  
    LPVOID lpvReserved,  
    LPSECURITYCREDENTIALS* lppCredentials  
);
```

The **ImapCreateSecurityCredentials** function creates a **SECURITYCREDENTIALS** structure.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is

	supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A pointer to a string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A pointer to a string which specifies the certificate owner's password. A value of NULL specifies

that no password is required. This parameter is only used if a PKCS12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A pointer to a string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

lpvReserved

Pointer reserved for future use. Set it to NULL when using this function.

lppCredentials

Pointer to an [LPSECURITYCREDENTIALS](#) pointer. The memory for the credentials structure will be allocated by this function and must be released by calling the **ImapDeleteSecurityCredentials** function when it is no longer needed. The pointer value must be set to NULL before the function is called. It is important to note that this is a pointer to a pointer variable, not a pointer to the SECURITYCREDENTIALS structure itself.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **ImapGetLastError**.

Remarks

The structure that is created by this function may be used as client credentials when establishing a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can

be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **ImapAsyncConnect** or **ImapConnect** functions.

Example

```
LPSECURITYCREDENTIALS lpSecCred = NULL;
ImapCreateSecurityCredentials(SEcurity_PROTOCOL_DEFAULT,
                             0,
                             NULL,
                             NULL,
                             lpszCertStore,
                             lpszCertName,
                             NULL,
                             &lpSecCred);

hClient = ImapConnect(lpszHostName,
                     IMAP_PORT_SECURE,
                     IMAP_TIMEOUT,
                     IMAP_OPTION_SECURE,
                     lpSecCred);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapAsyncConnect](#), [ImapConnect](#), [ImapDeleteSecurityCredentials](#), [ImapGetSecurityInformation](#), [SECURITYCREDENTIALS](#)

ImapDeleteMailbox Function

```
INT WINAPI ImapDeleteMailbox(  
    HCLIENT hClient,  
    LPCTSTR lpszMailbox  
);
```

The **ImapDeleteMailbox** function deletes a mailbox on the server.

Parameters

hClient

Handle to the client session.

lpszMailbox

A pointer to a string which specifies the mailbox to be deleted.

Return Value

If the function succeeds, it returns value of zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

A mailbox cannot be deleted if it contains inferior hierarchical names and has the IMAP_FLAG_NOSELECT attribute. On most systems this is the case when the mailbox name references a directory on the server, and that directory contains other subdirectories or mailboxes. To remove the mailbox, you must first delete any child mailboxes that exist.

If the mailbox that is deleted is the currently selected mailbox, it will be automatically unselected and any messages marked for deletion will be expunged before the mailbox is removed. If the delete operation fails, the client will remain in an unselected state until either the **ImapExamineMailbox** or **ImapSelectMailbox** function is called.

The special mailbox name INBOX is reserved, and cannot be deleted.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapCreateMailbox](#), [ImapGetFirstMailbox](#), [ImapGetNextMailbox](#), [ImapRenameMailbox](#)

ImapDeleteMessage Function

```
INT WINAPI ImapDeleteMessage(  
    HCLIENT hClient,  
    UINT nMessageId  
);
```

The **ImapDeleteMessage** function marks the specified message for deletion from the current mailbox.

Parameters

hClient

Handle to the client session.

nMessage

Number of message to delete from the server. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

This function only marks the message for deletion. The message is not actually deleted until the mailbox is expunged or another mailbox is selected. This function will return an error if the current mailbox is in read-only mode, such as if it was selected using the **ImapExamineMailbox** function.

It is important to note that unlike the POP3 protocol, a message that is marked for deletion is still accessible on the IMAP server until the mailbox is expunged. This means, for example, that a deleted message can still be retrieved using the **ImapGetMessage** function.

To determine if a message has been marked for deletion, use the **ImapGetMessageFlags** function and check if the IMAP_FLAG_DELETED bit flag has been set. To list all of the deleted messages in the current mailbox, use the **ImapGetDeletedMessages** function.

To remove the deletion flag from the message, use the **ImapUndeleteMessage** function. To prevent all messages in the current mailbox from being expunged, use the **ImapReselectMailbox** function to reset the current mailbox state.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

See Also

[ImapGetDeletedMessages](#), [ImapGetMessage](#), [ImapGetMessageCount](#), [ImapGetMessageFlags](#), [ImapReselectMailbox](#), [ImapUndeleteMessage](#), [ImapUnselectMailbox](#)

ImapDeleteSecurityCredentials Function

```
VOID WINAPI ImapDeleteSecurityCredentials(  
    LPSECURITYCREDENTIALS* LppCredentials  
);
```

The **ImapDeleteSecurityCredentials** function deletes an existing **SECURITYCREDENTIALS** structure.

Parameters

lppCredentials

Pointer to an [LPSECURITYCREDENTIALS](#) pointer. On exit from the function, the pointer will be NULL.

Return Value

None.

Example

```
if (lpSecCred)  
    ImapDeleteSecurityCredentials(&lpSecCred);  
  
ImapUninitialize();
```

Remarks

This function can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmapi10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapCreateSecurityCredentials](#), [ImapUninitialize](#)

ImapDisableEvents Function

```
INT WINAPI ImapDisableEvents(  
    HCLIENT hClient  
);
```

The **ImapDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapDisableEvents** function is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this function has been called.

This function is automatically called if the client has event notification enabled, and the **ImapDisconnect** function is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

See Also

[ImapAsyncConnect](#), [ImapEnableEvents](#), [ImapRegisterEvent](#)

ImapDisableTrace Function

```
BOOL WINAPI ImapDisableTrace();
```

The **ImapDisableTrace** function disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **ImapGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmavp10.lib

See Also

[ImapEnableTrace](#)

ImapDisconnect Function

```
INT WINAPI ImapDisconnect(  
    HCLIENT hClient  
);
```

The **ImapDisconnect** function terminates the connection with the server, releasing the memory allocated for the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

If a mailbox is selected at the time that **ImapDisconnect** is called, the current mailbox will be unselected and any messages that are marked for deletion will be expunged. To prevent any deleted messages from being removed from the mailbox, use the **ImapUnselectMailbox** function to unselect the current mailbox prior to disconnecting from the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[ImapAsyncConnect](#), [ImapConnect](#), [ImapUninitialize](#), [ImapUnselectMailbox](#)

ImapEnableEvents Function

```
INT WINAPI ImapEnableEvents(  
    HCLIENT hClient,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **ImapDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **ImapRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. To avoid conflict with standard Windows messages, this value must be greater than WM_USER (1024) or an error will be returned. If the *hEventWnd* parameter is NULL, this value should be WM_NULL.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapEnableEvents** function is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer. The client must create a window message handler, which processes the various events. The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event identifier, and the high word will contain any error code. If no error has occurred, the high word will have a value of zero. The following events may be generated:

Constant	Description
IMAP_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
IMAP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
IMAP_EVENT_READ	Data is available to read by the client. No additional messages will

	be posted until the client has read at least some of the data. This event is only generated if the calling process is in asynchronous mode.
IMAP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
IMAP_EVENT_TIMEOUT	The client has timed out while waiting for a response from the server. Note that under some circumstances this event can be generated for a non-blocking connection, such as when the client is establishing a secure connection.
IMAP_EVENT_CANCEL	The client has canceled the current operation.
IMAP_EVENT_COMMAND	The client has processed a command that was sent to the server. The result code and result string can be used to determine if the response to the command. The high word of the IParam parameter should be checked, since this notification message will also be posed if the command cannot be executed.
IMAP_EVENT_PROGRESS	This event notification is sent periodically during lengthy blocking operations, such as retrieving a complete message from the server.

It is not required that the client be placed in asynchronous (non-blocking) mode in order to receive event notifications, except for the connect, disconnect, read and write events. To disable event notification, call the **ImapDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[ImapAsyncConnect](#), [ImapDisableEvents](#), [ImapRegisterEvent](#)

ImapEnableTrace Function

```
BOOL WINAPI ImapEnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **ImapEnableTrace** function enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

Name of the trace log file. If this parameter is NULL or empty, the file CSTRACE.LOG is used. The directory for CSTRACE.LOG is given by the TEMP environment variable, if it is defined; otherwise, the directory given by the TMP environment variable is used, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Constant	Description
0	TRACE_DEFAULT	All function calls and return values are written to the trace file. The actual data being sent or received will not be logged. This is the default value.
1	TRACE_ERROR	Only those function calls which fail are recorded in the trace file. Those errors which are not fatal and only indicate a warning will not be logged.
2	TRACE_WARNING	Only those function calls which fail, or return values which indicate a warning, are recorded in the trace file.
4	TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **ImapGetLastError**.

Remarks

When trace logging is enabled, the logfile is opened, appended to and closed for each socket function call. Using the same logfile name, you can do the same in your application to add additional information to the logfile if needed. This can provide an application-level context for the entries made by the library. Make sure that the logfile is closed after the data has been written.

The TRACE_HEXDUMP option can produce very large logfiles, since all data that is being sent and received by the application is logged. To reduce the size of the file, you can enable and disable logging around limited sections of code that you wish to analyze.

All of the SocketTools networking components that use the Windows Sockets API support logging. If you are using multiple components, you only need to enable tracing once in your application or once per thread in a multithreaded application.

To redistribute an application that includes logging functionality, the **cstrcv10.dll** library must be included as part of the installation package. This library provides the trace logging features, and if it is not available the **ImapEnableTrace** function will fail. Note that the trace logging library is a standard Windows DLL and does not need to be registered, it only needs to be redistributed with your application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmav10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapDisableTrace](#)

ImapEnumMessages Function

```
INT WINAPI ImapEnumMessages(  
    HCLIENT hClient,  
    UINT nFirstMessageId,  
    UINT nLastMessageId,  
    DWORD dwMessageFlags,  
    DWORD dwReserved,  
    LPIMAPMESSAGE lpMessageList,  
    INT nMaxMessages  
);
```

The **ImapEnumMessages** function enumerates the messages in the current mailbox, populating an array of IMAPMESSAGE structures which contain information about each message.

Parameters

hClient

Handle to the client session.

nFirstMessageId

An unsigned integer value which specifies the first message to enumerate. This value must be greater than zero and specify a valid message identifier. The first message in the mailbox has a value of one.

nLastMessageId

An unsigned integer value which specifies the last message to enumerate. This value must be greater than or equal to the value of the *nFirstMessageId* parameter. A special value of 0xFFFFFFFF can be used to specify the last message in the mailbox.

dwMessageFlags

This parameter is used to determine which messages are enumerated. If the value is zero, then all applicable messages will be enumerated. If the value is non-zero, only those messages which have at least one of the specified flags will be returned. More than one flag can be specified by using a bitwise operator. Valid message flags are:

Constant	Description
IMAP_FLAG_ANSWERED	Return only those messages which have been answered.
IMAP_FLAG_DELETED	Return only those messages which have been marked for deletion.
IMAP_FLAG_DRAFT	Return only those messages which have been marked as draft copies.
IMAP_FLAG_URGENT	Return only those messages which have been flagged for urgent or special attention.
IMAP_FLAG_RECENT	Return only those messages which have been recently added to the mailbox.
IMAP_FLAG_SEEN	Return only those messages which have been read.

dwReserved

A reserved parameter. This value should always be set to zero.

lpMessageList

A pointer to an array of [IMAPMESSAGE](#) structures which will contain information about each of the messages returned by the server. This parameter cannot be NULL.

nMaxMessages

An integer value which specifies the maximum size of the IMAPMESSAGE array that was passed to the function. This value must be at least one.

Return Value

If the function succeeds, the return value is the number of messages that were enumerated. If no messages match the specified criteria, the function will return a value of zero. If an error is encountered, the function returns IMAP_ERROR. To get extended error information, call [ImapGetLastError](#).

Remarks

If the message UID is being stored locally by the client to identify the message over multiple sessions, it must also store the mailbox UID. Only the combination of the mailbox name, mailbox UID and message UID can be used to uniquely identify a given message on the server. Although IMAP server implementations are encouraged to maintain persistent message UIDs, they are not required to do so and those values may change if the mailbox UID changes.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[ImapGetMessage](#), [ImapGetMessageCount](#), [ImapGetMessageFlags](#), [ImapGetMessageHeaders](#), [ImapGetMessageSize](#), [ImapGetMessageUid](#)

ImapEventProc Function

```
VOID CALLBACK ImapEventProc(  
    HCLIENT hClient,  
    UINT nEvent,  
    DWORD dwError,  
    DWORD_PTR dwParam  
);
```

The **ImapEventProc** function is an application-defined callback function that processes events generated by the client.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **ImapRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **ImapRegisterEvent** function. The **ImapEventProc** function is a placeholder for the application-defined function name.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmav10.lib

See Also

[ImapDisableEvents](#), [ImapEnableEvents](#), [ImapFreezeEvents](#), [ImapRegisterEvent](#)

ImapExamineMailbox Function

```
INT WINAPI ImapExamineMailbox(  
    HCLIENT hClient,  
    LPCTSTR lpszMailbox,  
    LPIMAPMAILBOX lpMailboxInfo  
);
```

The **ImapExamineMailbox** function selects the specified mailbox for read-only access.

Parameters

hClient

Handle to the client session.

lpszMailbox

A pointer to a string which specifies the new mailbox to be examined.

lpMailboxInfo

A pointer to an [IMAPMAILBOX](#) structure which contains information about the mailbox when the function returns. This parameter may be NULL if the caller does not require any information about the mailbox.

Return Value

If the function succeeds, it returns zero. If an error occurs, the function returns `IMAP_ERROR`. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapExamineMailbox** function is used to select a mailbox in read-only mode. Messages can be read, but they cannot be modified or deleted from the mailbox and new messages will not lose their status as new messages if they are accessed.

If the client has a different mailbox currently selected, that mailbox will be closed and any messages marked for deletion will be expunged. To prevent deleted messages from being removed from the previous mailbox, use the **ImapUnselectMailbox** function prior to examining the new mailbox.

If an application wishes to update the information returned in the `IMAPMAILBOX` structure for the current mailbox, simply call **ImapExamineMailbox** again with the same mailbox name.

The special case-insensitive mailbox name `INBOX` is used for new messages. Other mailbox names may or may not be case-sensitive depending on the IMAP server's operating system and implementation.

To access a mailbox in read-write mode, use the **ImapSelectMailbox** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmapi10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapDeleteMailbox](#), [ImapGetFirstMailbox](#), [ImapGetMailboxStatus](#), [ImapGetNextMailbox](#),

ImapExpungeMailbox Function

```
INT WINAPI ImapExpungeMailbox(  
    HCLIENT hClient,  
    LPIMAPMAILBOX lpMailboxInfo  
);
```

The **ImapExpungeMailbox** function removes all messages marked for deletion from the current mailbox.

Parameters

hClient

Handle to the client session.

lpMailboxInfo

A pointer to an [IMAPMAILBOX](#) structure which contains information about the mailbox when the function returns. This parameter may be NULL if the caller does not require any information about the mailbox.

Return Value

If the function succeeds, it returns zero. If an error occurs, the function returns `IMAP_ERROR`. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapExpungeMailbox** function causes all messages marked for deletion to be removed from the mailbox. Note that this can cause the mailbox's UID to change, and potentially invalidate the current message UIDs. It is recommended that applications use the information returned in the `IMAPMAILBOX` structure to update any internal state information stored on the local system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmapi10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapCreateMailbox](#), [ImapExamineMailbox](#), [ImapGetFirstMailbox](#), [ImapGetNextMailbox](#), [ImapRenameMailbox](#), [ImapSelectMailbox](#)

ImapFreezeEvents Function

```
INT WINAPI ImapFreezeEvents(  
    HCLIENT hClient,  
    BOOL bFreeze  
);
```

The **ImapFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four events of the same type occur, once event handling is resumed only one of those events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

See Also

[ImapDisableEvents](#), [ImapEnableEvents](#), [ImapRegisterEvent](#)

ImapGetCapability Function

```
INT WINAPI ImapGetCapability(  
    HCLIENT hClient,  
    LPTSTR lpszCapability,  
    INT nMaxLength  
);
```

The **ImapGetCapability** function returns a string which identifies the capabilities of the IMAP server.

Parameters

hClient

Handle to the client session.

lpszCapability

A pointer to a null terminated string buffer that will contain one or more tokens separated by spaces which identify the capabilities of the IMAP server.

nMaxLength

The maximum length of the capability string, including the terminating null character.

Return Value

If the function succeeds, it returns the length of the capability string, not including the terminating null character. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

This function returns a string that contains one or more tokens separated by whitespace. Each token identifies a capability of the server. The following table lists some of the common capabilities:

Capability	Description
ACL	The RFC 2086 ACL extension
BINARY	The RFC 3516 binary content extension
CHILDREN	The RFC 3348 child mailbox extension
ID	The RFC 2971 ID extension
IDLE	The RFC 2177 IDLE extension
LOGINDISABLED	The RFC 2595 TLS/SSL extension
LOGINREFERRALS	The RFC 2221 login referrals extension
MAILBOXREFERRALS	The RFC 2193 mailbox referrals extension
MULTIAPPEND	The RFC 3501 MULTIAPPEND extension
NAMESPACE	The RFC 2342 namespace Extension
QUOTA	The RFC 2087 QUOTA extension
STARTTLS	The RFC 2595 TLS/SSL extension
UNSELECT	The RFC 3691 UNSELECT extension

Additional capabilities may be supported by your server. Note that experimental or custom capabilities are always prefixed with the letter X. A list of standard IMAP capabilities is maintained by the Internet Assigned Numbers Authority (IANA) at www.iana.org.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmapi10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapConnect](#), [ImapDisconnect](#)

ImapGetCurrentMailbox Function

```
INT WINAPI ImapGetCurrentMailbox(  
    HCLIENT hClient,  
    LPTSTR lpszMailbox,  
    INT nMaxLength  
);
```

The **ImapGetCurrentMailbox** function returns the name of the current mailbox that has been selected.

Parameters

hClient

Handle to the client session.

lpszMailbox

A pointer to a null terminated string buffer that will contain the current mailbox name.

nMaxLength

The maximum length of the mailbox string, including the terminating null character.

Return Value

If the function succeeds, it returns the length of the current mailbox name, not including the terminating null character. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapExamineMailbox](#), [ImapSelectMailbox](#), [ImapUnselectMailbox](#)

ImapGetDeletedMessages Function

```
INT WINAPI ImapGetDeletedMessages(  
    HCLIENT hClient,  
    UINT* LpnMessageIds,  
    INT nMaxMessages  
);
```

The **ImapGetDeletedMessages** function returns the message identifiers for those messages that have been marked for deletion in the current mailbox.

Parameters

hClient

Handle to the client session.

lpnMessageIds

A pointer to an array of unsigned integers that will contain the message identifiers of those messages which have been marked for deletion in the current mailbox. This parameter may be NULL, in which case the function will return the number of deleted messages but will not return their identifiers.

nMaxMessages

The maximum number of message identifiers that may be stored in the *lpnMessageIds* array. If the *lpnMessageIds* parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the number of messages marked for deletion in the current mailbox. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The message identifiers returned by this function are only valid until the mailbox is expunged or another mailbox is selected.

To remove the deleted flag from a message, use the **ImapUndeleteMessage** function. To prevent all messages in the current mailbox from being expunged, use the **ImapReselectMailbox** function to reset the current mailbox state.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

See Also

[ImapGetMessageCount](#), [ImapGetMessageFlags](#), [ImapGetNewMessages](#), [ImapGetUnseenMessages](#), [ImapSearchMailbox](#)

ImapGetErrorString Function

```
INT WINAPI ImapGetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT nMaxLength  
);
```

The **ImapGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **ImapGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description should be returned. If this value is zero, then the description of the last error will be returned. If the last error code is zero, indicating no error, then this function will return zero.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. It is recommended that this buffer be at least 128 characters in length. If a NULL pointer is specified, then no message will be returned but the function will return the length of the error string, not including the terminating null byte.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is 0, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapGetLastError](#), [ImapGetResultCode](#), [ImapGetResultString](#), [ImapSetLastError](#)

ImapGetFirstMailbox Function

```
INT WINAPI ImapGetFirstMailbox(  
    HCLIENT hClient,  
    LPCTSTR lpszReference,  
    LPCTSTR lpszWildcard,  
    DWORD dwOptions,  
    LPVOID lpvReserved,  
    LPTSTR lpszMailbox,  
    INT nMaxLength  
    LPDWORD lpdwFlags  
);
```

The **ImapGetFirstMailbox** function returns the name of the first matching mailbox.

Parameters

hClient

Handle to the client session.

lpszReference

A pointer to a string which specifies the reference name. An empty string or NULL pointer specifies that the default mailbox hierarchy for the current user is returned. If the reference name is provided, this must be the name of a mailbox or a level of the mailbox hierarchy which provides the context in which the mailbox name is interpreted.

lpszWildcard

A pointer to a null terminated string which specifies the mailbox name to match. The wildcard character "*" may be used to match any portion of the mailbox hierarchy, including the delimiter. The wildcard character "%" matches any portion of the mailbox name, but does not match the mailbox delimiter. An empty string or NULL pointer specifies that all mailboxes in the context of the *lpszReference* parameter should be returned.

dwOptions

Specifies one or more options which controls how mailboxes are returned by the function. The options are bit flags which may be combined using a bitwise operator. One or more of the following values may be used:

Constant	Description
IMAP_LIST_DEFAULT	This option specifies that all regular, selectable mailboxes should be returned.
IMAP_LIST_SUBSCRIBED	This option specifies that only subscribed mailboxes should be returned.
IMAP_LIST_FOLDERS	This option specifies that non-selectable mailbox folders should also be returned.
IMAP_LIST_HIDDEN	This option specifies that hidden mailboxes should be returned.
IMAP_LIST_INFERIOR	This option specifies that inferior mailboxes should be returned if an explicit wildcard mask is not specified.

lpvReserved

A reserved parameter. This value should always be set to NULL.

lpszMailbox

A pointer to a string buffer which will contain the first matching mailbox. This parameter cannot be NULL. A minimum buffer size of at least 128 character is recommended.

nMaxLength

Specifies the maximum length of the string buffer. The maximum length of the buffer should be large enough to accommodate most path names on the IMAP server.

lpdwFlags

A pointer to an unsigned integer which will contain the mailbox flags for the first matching mailbox. This parameter may be NULL, in which case the mailbox flags are not returned. Otherwise, one or more of the following bit flags may be returned:

Constant	Description
IMAP_FLAG_NOINFERIORS	The mailbox does not contain any sub-mailboxes. In the IMAP protocol, these are referred to as inferior hierarchical mailbox names.
IMAP_FLAG_NOSELECT	The mailbox cannot be selected or examined. This flag is typically used by servers to indicate that the mailbox name refers to a directory on the server, not a mailbox file.
IMAP_FLAG_MARKED	The mailbox is marked as being of interest to a client. If this flag is used, it typically means that the mailbox contains messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.
IMAP_FLAG_UNMARKED	The mailbox is marked as not being of interest to a client. If this flag is used, it typically means that the mailbox does not contain any messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.

Return Value

If the function succeeds, it returns the length of the mailbox name. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetFirstMailbox** function is used to begin enumerating the available mailboxes for the current user on the IMAP server. Subsequent mailbox names are returned by calling **ImapGetNextMailbox** until the function returns IMAP_ERROR with an error code of ST_ERROR_NO_MORE_MAILBOXES.

The function of the *lpszReference* and *lpszWildcard* parameters are implementation dependent and generally are tied to the underlying operating system. On a UNIX based system, it can be helpful to think of the reference name as the directory where mailbox folders are stored, and the mailbox name as the name to search for in that directory and any subdirectories, if applicable. If the reference name is an empty string or NULL pointer, this typically refers to the current user's home directory.

Generally speaking, a reference name should only be specified if you or the user of the application knows the directory structure on the IMAP server. Incorrectly using a reference name can have serious negative side-effects. For example, specifying a reference name of "/" on a UNIX based system could cause the IMAP server to return search every directory on the system for a matching mailbox name. Similarly, the IMAP server may be unable to distinguish between regular files in the user's home directory and mailboxes. For this reason, most IMAP clients require that the user specify the directory on the server where their mailboxes are stored. Typically this is subdirectory named "mail" or "Mail" under the user's home directory. For non-UNIX servers, the mailbox hierarchy may be represented differently, including a flat hierarchy.

Hidden mailboxes are those mailboxes which use the UNIX convention of the name beginning with a period. Therefore, a mailbox named ".secrets" would not normally be returned by the **ImapGetFirstMailbox** and **ImapGetNextMailbox** functions. The IMAP_LIST_HIDDEN option causes all mailboxes to be returned.

The IMAP_LIST_INFERIOR option will return inferior mailboxes (mailboxes located in folders or subdirectories) if a wildcard mask is not specified. If a wildcard mask is specified, this option has no effect and only those mailboxes which match the wildcard will be returned.

Subscribed mailboxes are those which were specified using the **ImapSubscribeMailbox** function. Marked mailboxes are typically those which have some special importance to the user.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapDeleteMailbox](#), [ImapGetMailboxStatus](#), [ImapGetNextMailbox](#), [ImapRenameMailbox](#), [ImapSelectMailbox](#)

ImapGetHeaderValue Function

```
INT WINAPI ImapGetHeaderValue(  
    HCLIENT hClient,  
    UINT nMessageId,  
    LPCTSTR LpszHeader,  
    LPTSTR LpszValue,  
    INT nMaxLength  
);
```

The **ImapGetHeaderValue** function returns the value of a header field in the specified message.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve header value from. This value must be greater than zero. The first message in the mailbox is message number one.

lpszHeader

Pointer to a string which specifies the message header to retrieve. The colon should not be included in this string.

lpszValue

Pointer to a string buffer that will contain the value of the specified message header.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, it returns the length of the header field value. If the header field is not present in the message, the function will return a value of zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetHeaderValue** function returns the value of header field from the specified message. This allows an application to be able to easily determine the value of a header such as the sender, or the subject of the message. Any header field, including non-standard extensions, may be returned by this function.

To return the value of a header field from a specific part of a multipart MIME message, use the **ImapGetHeaderValueEx** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapGetHeaderValueEx](#), [ImapGetMessageHeaders](#), [ImapGetMessageId](#), [ImapGetMessageSender](#)

ImapGetHeaderValueEx Function

```
INT WINAPI ImapGetHeaderValueEx(  
    HCLIENT hClient,  
    UINT nMessageId,  
    UINT nMessagePart,  
    LPVOID lpvReserved,  
    LPCTSTR lpszHeader,  
    LPTSTR lpszValue,  
    INT nMaxLength  
);
```

The **ImapGetHeaderValueEx** function returns the value of a header field from the specified message part.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve header value from. This value must be greater than zero. The first message in the mailbox is message number one.

nMessagePart

The message part that the header value will be retrieved from. Message parts start with a value of one. A value of zero specifies that the RFC822 header field for the message will be used.

lpvReserved

A reserved parameter. This value should always be NULL.

lpszHeader

Pointer to a string which specifies the message header to retrieve. The colon should not be included in this string.

lpszValue

Pointer to a string buffer that will contain the value of the specified message header.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, it returns the length of the header field value. If the header field is not present in the message, the function will return a value of zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetHeaderValueEx** function returns the value of a header field from the specified message part. This allows an application to be able to easily determine the value of a header such as the sender, or the subject of the message. Any header field, including non-standard extensions, may be returned by this function.

To determine if a message is a multipart MIME message, use the **ImapGetMessageParts** function. The return value specifies the number of parts in the message, with a value greater than

one indicating that it is a multipart message.

Note that unlike the SocketTools MIME API which considers the first message part to be zero, the IMAP protocol defines the first message part to be one.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapGetHeaderValue](#), [ImapGetMessageHeaders](#), [ImapGetMessageId](#), [ImapGetMessageSender](#)

ImapGetIdleThreadId Function

```
DWORD WINAPI ImapGetIdleThreadId(  
    HCLIENT hClient  
);
```

The **ImapGetIdleThreadId** function returns ID of the thread that is monitoring the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, it returns the ID of the thread that is checking for update notifications from the server. If there is no active thread monitoring the client session, the function will return zero.

Remarks

The worker thread that monitors the client connection in the background can terminate if an IMAP command is sent to the server, if the **ImapCancel** function is called or if the client disconnects from the server. The **ImapGetIdleThreadId** function enables the application to determine if this background thread is still active or not.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapIdle](#), [ImapIdleProc](#)

ImapGetLastError Function

```
DWORD WINAPI ImapGetLastError();
```

Parameters

None.

Return Value

The return value is the calling thread's last error code value. Functions set this value by calling the **ImapSetLastError** function. The return value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

You should call the **ImapGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **ImapSetLastError(0)** when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value error code such as FALSE, NULL, INVALID_CLIENT or IMAP_ERROR. Those functions which call **ImapSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[ImapGetErrorString](#), [ImapSetLastError](#)

ImapGetMailboxDelimiter Function

```
INT WINAPI ImapGetMailboxDelimiter(  
    HCLIENT hClient,  
    LPCTSTR lpszMailbox,  
    LPTSTR lpszDelimiter,  
    INT nMaxLength  
);
```

The **ImapGetMailboxDelimiter** function returns the hierarchical path delimiter used for the specified mailbox.

Parameters

hClient

Handle to the client session.

lpszMailbox

A pointer to a null terminated string which specifies the mailbox name. This parameter may be NULL or an empty string, in which case the default delimiter will be returned.

lpszDelimiter

A pointer to a null terminated string buffer that will contain the mailbox delimiter.

nMaxLength

The maximum length of the delimiter string, including the terminating null character.

Return Value

If the function succeeds, it returns the length of the delimiter for the specified mailbox, not including the terminating null character. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

If the IMAP server supports multiple levels of mailboxes, then a special character or sequence of characters are used as delimiters between different levels of the mailbox hierarchy. On most systems, including most UNIX and Windows platforms, the delimiter is the forward slash "/" character.

It is possible that an IMAP server may only support a flat namespace, in which case this function will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapExamineMailbox](#), [ImapSelectMailbox](#), [ImapUnselectMailbox](#)

ImapGetMailboxSize Function

```
DWORD WINAPI ImapGetMailboxSize(  
    HCLIENT hClient,  
    LPCTSTR lpszMailbox  
);
```

The **ImapGetMailboxSize** function returns the size of the specified mailbox.

Parameters

hClient

Handle to the client session.

lpszMailbox

A pointer to a string which specifies the mailbox name.

Return Value

If the function succeeds, it returns the size of the mailbox. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMailboxSize** function may require a significant amount of time to calculate the mailbox size if there are a large number of messages in the mailbox. If the specified mailbox is not currently selected, then the current mailbox is unselected, the new mailbox is selected and the size calculated, and then the original mailbox is re-selected. This will have the side-effect of causing any messages marked for deletion to be expunged from the mailbox.

Because it can potentially result in long delays, it is not recommended that an application calculate the mailbox size unless it is absolutely necessary.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapCreateMailbox](#), [ImapGetFirstMailbox](#), [ImapGetNextMailbox](#), [ImapRenameMailbox](#)

ImapGetMailboxStatus Function

```
INT WINAPI ImapGetMailboxStatus(  
    HCLIENT hClient,  
    LPCTSTR lpszMailbox,  
    LPIMAPMAILBOXSTATUS lpMailboxStatus  
);
```

The **ImapGetMailboxStatus** function returns status information about the specified mailbox.

Parameters

hClient

Handle to the client session.

lpszMailbox

A pointer to a null terminated string that specifies the mailbox to return information about.

lpMailboxStatus

A pointer to an [IMAPMAILBOXSTATUS](#) structure which contains status information about the specified mailbox.

Return Value

If the function succeeds, it returns zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMailboxStatus** function enables an application to obtain status information about a mailbox without having to select another mailbox or open a second connection to the IMAP server to examine the mailbox. The information returned is a subset of the information returned when a mailbox is selected.

Note that obtaining status information for a mailbox may be a slow operation. It may require that server open the mailbox in read-only mode internally in order to obtain some of the status information. For this reason, this function should not be used to check for new messages; use the **ImapCheckMailbox** function instead.

Some IMAP servers may return an error if you attempt to obtain status information about the currently selected mailbox. The protocol standard states that clients should not use this method on the currently selected mailbox, and should instead use the information returned by the **ImapSelectMailbox** or **ImapExamineMailbox** functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapCheckMailbox](#), [ImapExamineMailbox](#), [ImapSelectMailbox](#)

ImapGetMailboxUID Function

```
DWORD WINAPI ImapGetMailboxUID(  
    HCLIENT hClient,  
    LPCTSTR lpszMailbox  
);
```

The **ImapGetMailboxUID** function returns the unique identifier for the specified mailbox.

Parameters

hClient

Handle to the client session.

lpszMailbox

A pointer to a null terminated string that specifies the mailbox name.

Return Value

If the function succeeds, it returns a non-zero value. If no unique identifier is assigned to the mailbox, the function will return zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMailboxUID** function returns an unsigned 32-bit value which uniquely identifies the mailbox and corresponds to the UIDVALIDITY value returned by the IMAP server. The actual value is determined by the server and should be considered opaque. The protocol specification requires that a mailbox's UID must not change unless the mailbox contents are modified or existing messages in the mailbox have been assigned new UIDs.

An application can use the mailbox UID value in combination with the message UID in order to uniquely identify a message on the server. However, the application must take into consideration that the IMAP server can reassign new message UIDs when the mailbox is modified. If the mailbox and message UIDs are being stored on the local system to track what messages have been retrieved from the server, the application must check the UID of the mailbox whenever it is selected. If the mailbox UID has changed, this means that the UIDs for the messages in that mailbox may have changed. The client should resynchronize with the server, and update it's local copy of that mailbox.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapGetCurrentMailbox](#), [ImapGetMailboxStatus](#), [ImapGetMessageUid](#)

ImapGetMessage Function

```
INT WINAPI ImapGetMessage(  
    HCLIENT hClient,  
    UINT nMessageId,  
    UINT nMessagePart,  
    LPVOID lpvMessage,  
    LPDWORD lpdwLength,  
    DWORD dwOptions  
);
```

The **ImapGetMessage** function retrieves a message from the server.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve from the server. This value must be greater than zero. The first message in the mailbox is message number one.

nMessagePart

The message part that will be retrieved. A value of zero specifies that the complete message should be returned. If the message is a multipart MIME message, message parts start with a value of one.

lpvMessage

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvMessage* parameter. If the *lpvMessage* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the message that was downloaded.

dwOptions

An unsigned integer value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
IMAP_SECTION_DEFAULT	All headers and the complete body of the specified message or message part are to be retrieved. The client application is responsible for parsing the header block. If the message is a MIME multipart message and the complete message is returned, the application is responsible for parsing the individual message parts if necessary.
IMAP_SECTION_HEADER	All headers for the specified message or message part are to be retrieved. The client application is responsible for parsing the header block.

IMAP_SECTION_MIMEHEADER	The MIME headers for the specified message or message are to be retrieved. Only those header fields which are used in MIME messages, such as Content-Type will be returned to the client. This is typically useful when processing the header for a multipart message which contains file attachments. The client application is responsible for parsing the header block.
IMAP_SECTION_BODY	The body of the specified message or message part is to be retrieved. For a MIME formatted message, this may include data that is uuencoded or base64 encoded. The application is responsible for decoding this data.
IMAP_SECTION_PREVIEW	The message header or body is being previewed and should not be marked as read by the server. This prevents the message from having the IMAP_FLAG_SEEN flag from being automatically set when the message data is retrieved.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMessage** function is used to retrieve a message from the server and copy it into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the message. In this case, the *lpvMessage* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvMessage* parameter point to a global memory handle which will contain the message data when the function returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur.

This function will cause the current thread to block until the transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the IMAP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **ImapEnableEvents**, or by registering a callback function using the **ImapRegisterEvent** function.

To determine if a message is a multipart MIME message, use the **ImapGetMessageParts** function. The return value specifies the number of parts in the message, with a value greater than one indicating that it is a multipart message. Combining the IMAP_SECTION_HEADER and IMAP_SECTION_BODY options will only return the header and body for the specified message if the *nMessagePart* parameter is zero. Due to a limitation of the IMAP FETCH command, if a message part is specified then only the body of that message part will be returned.

Note that unlike the SocketTools MIME API which considers the first message part to be zero, the IMAP protocol defines the first message part to be one.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[ImapGetMessageCount](#), [ImapGetMessageHeaders](#), [ImapGetMessageParts](#), [ImapStoreMessage](#)

ImapGetMessageCount Function

```
INT WINAPI ImapGetMessageCount(  
    HCLIENT hClient  
);
```

The **ImapGetMessageCount** function returns the number of messages that are available in the currently selected mailbox.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, it returns the number of messages in the currently selected mailbox. If no messages are available, this function will return zero. If the function fails, the return value is `IMAP_ERROR`. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMessageCount** function does not distinguish between messages that are marked for deletion and those which are not. This is because messages that are marked for deletion on an IMAP server can still be accessed until the mailbox is expunged or unselected. This differs from the POP3 protocol, where messages cannot be accessed once they have been marked for deletion.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmapi10.lib`

See Also

[ImapDeleteMessage](#), [ImapGetHeaderValue](#), [ImapGetMessage](#), [ImapGetMessageCountEx](#), [ImapGetMessageHeaders](#), [ImapStoreMessage](#)

ImapGetMessageCountEx Function

```
INT WINAPI ImapGetMessageCountEx(  
    HCLIENT hClient,  
    UINT *LpnLastMessage,  
    DWORD *LpdwMailboxSize  
);
```

The **ImapGetMessageCountEx** function returns the number of messages that are available in the currently selected mailbox, and optionally the size of the mailbox in bytes.

Parameters

hClient

Handle to the client session.

lpnLastMessage

Address of a variable that receives the number of the last valid message in the mailbox. If a NULL value is specified, this argument is ignored.

lpdwMailboxSize

Address of a variable that receives the current size of the mailbox. If a NULL value is specified, this argument is ignored.

Return Value

If the function succeeds, it returns the number of messages that are currently available. If no messages are available, this function will return zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMessageCountEx** function is provided for compatibility with the POP3 API. The *lpnLastMessage* parameter will always contain the same value returned by the function since there is no distinction between the message count and the last available message. This is because messages that are marked for deletion on an IMAP server can still be accessed until the mailbox is expunged or unselected. This differs from the POP3 protocol, where messages cannot be accessed once they have been marked for deletion.

If the *lpdwMailboxSize* parameter is specified, this function will call **ImapGetMailboxSize** to determine the size of the currently selected mailbox. Unlike the POP3 protocol, calculating the mailbox size may require a significant amount of time if there are a large number of messages in the mailbox. It is not recommended that an application request the mailbox size unless it is absolutely necessary.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapDeleteMessage](#), [ImapGetHeaderValue](#), [ImapGetMailboxSize](#), [ImapGetMessage](#), [ImapGetMessageHeaders](#), [ImapStoreMessage](#)

ImapGetMessageFlags Function

```
BOOL WINAPI ImapGetMessageFlags(  
    HCLIENT hClient,  
    UINT nMessageId,  
    LPDWORD lpdwMessageFlags  
);
```

The **ImapGetMessageFlags** function returns the message flags for the specified message.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to obtain the message flags for. This value must be greater than zero. The first message in the mailbox is message number one.

lpdwMessageFlags

Pointer to an unsigned integer that will contain the message flags for the specified message. The value may be zero, or one or more of the following values:

Constant	Description
IMAP_FLAG_ANSWERED	The message has been answered.
IMAP_FLAG_DELETED	The message has been marked for deletion.
IMAP_FLAG_DRAFT	The message has not been completed and is marked as a draft copy.
IMAP_FLAG_URGENT	The message has been flagged for urgent or special attention.
IMAP_FLAG_RECENT	The message has been added to the mailbox recently.
IMAP_FLAG_SEEN	The message has been read.

Return Value

If the function succeeds, the return value is non-zero and the *lpdwMessageFlags* parameter contains the status flags for the specified message. If the function fails, the return value is zero. To get extended error information, call **ImapGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

See Also

[ImapDeleteMessage](#), [ImapGetMessageCount](#), [ImapSetMessageFlags](#)

ImapGetMessageHeaders Function

```
INT WINAPI ImapGetMessageHeaders(  
    HCLIENT hClient,  
    UINT nMessageId,  
    LPVOID lpvHeaders,  
    LPDWORD lpdwLength  
);
```

The **ImapGetMessageHeaders** function retrieves the headers for the specified message from the server.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve from the server. This value must be greater than zero. The first message in the mailbox is message number one.

lpvHeaders

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvHeaders* parameter. If the *lpvHeaders* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the message that was downloaded.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMessageHeaders** function is used to retrieve a message header block from the server and copy it into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the *lpvHeaders* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvHeaders* parameter point to a global memory handle which will contain the file data when the function returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the IMAP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **ImapEnableEvents**, or by registering a callback function using

the **ImapRegisterEvent** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[ImapGetMessage](#), [ImapGetMessageCount](#), [ImapGetMessageParts](#), [ImapStoreMessage](#)

ImapGetMessageHeadersEx Function

```
INT WINAPI ImapGetMessageHeadersEx(  
    HCLIENT hClient,  
    UINT nMessageId,  
    DWORD dwReserved,  
    LPDWORD lpdwMessageUID,  
    LPDWORD lpdwMessageFlags,  
    LPVOID lpvHeaders,  
    LPDWORD lpdwLength  
);
```

The **ImapGetMessageHeadersEx** function retrieves the headers, unique ID and flags for the specified message.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve from the server. This value must be greater than zero. The first message in the mailbox is message number one.

dwReserved

A reserved parameter, this value should always be zero.

lpdwMessageUID

A pointer to an unsigned integer which will contain the unique ID assigned to the message by the server when the function returns. This parameter may be NULL if this information is not required.

lpdwMessageFlags

A pointer to an unsigned integer which will contain the message flags when the function returns. This parameter may be NULL if this information is not required, otherwise the value may be zero or one or more of the following bitflags:

Constant	Description
IMAP_FLAG_ANSWERED	The message has been answered.
IMAP_FLAG_DELETED	The message has been marked for deletion.
IMAP_FLAG_DRAFT	The message has not been completed and is marked as a draft copy.
IMAP_FLAG_URGENT	The message has been flagged for urgent or special attention.
IMAP_FLAG_RECENT	The message has been added to the mailbox recently.
IMAP_FLAG_SEEN	The message has been read.

lpvHeaders

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvHeaders* parameter. If the *lpvHeaders* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the message that was downloaded.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMessageHeadersEx** function is used to retrieve a message header block from the server and copy it into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the *lpvHeaders* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvHeaders* parameter point to a global memory handle which will contain the file data when the function returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur.

If the *lpdwMessageUID* and/or *lpdwMessageFlags* parameters are NULL, they will be ignored. Otherwise, the function will always initialize the values to zero and only sets them if the header block for the message can be retrieved.

This function will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the IMAP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **ImapEnableEvents**, or by registering a callback function using the **ImapRegisterEvent** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: csmavp10.lib

See Also

[ImapGetMessage](#), [ImapGetMessageCount](#), [ImapGetMessageFlags](#), [ImapGetMessageParts](#), [ImapStoreMessage](#)

ImapGetMessageId Function

```
INT WINAPI ImapGetMessageId(  
    HCLIENT hClient,  
    UINT nMessageId,  
    LPTSTR lpszMessageId,  
    INT nMaxLength  
);
```

The **ImapGetMessageId** function returns the message identifier string for the specified message.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve the unique identifier for. This value must be greater than zero. The first message in the mailbox is message number one.

lpszMessageId

Address of a string buffer to receive the message identifier. This should be at least 64 bytes in length.

nMaxLength

The maximum length of the string buffer.

Return Value

If the function succeeds, the return value is the length of the unique identifier string. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMessageId** function returns the message identifier from the Message-ID header of the specified message. The returned value is a string which can be used to identify a specific message, regardless if the message is moved to a different mailbox.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapGetHeaderValue](#), [ImapGetMessage](#), [ImapGetMessageHeaders](#), [ImapGetMessageSender](#)

ImapGetMessageParts Function

```
INT WINAPI ImapGetMessageParts(  
    HCLIENT hClient,  
    UINT nMessageId  
);
```

The **ImapGetMessageParts** function returns the number of parts in a MIME multipart message on the server.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve the part count for. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the function succeeds, the return value is the number of parts in the specified message. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMessageParts** function can be used to determine if a message on the server contains multiple parts. A multipart MIME message typically contains file attachments or multiple representations of the message, such as a version of the message in plain text and another using HTML markup.

If the function returns a value of one, then the message does not contain multiple parts and is a standard RFC822 formatted message. A value greater than one indicates that the message does have multiple parts. The **ImapGetMessageEx** function may be used to retrieve the data for a specific part of the message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

See Also

[ImapGetHeaderValue](#), [ImapGetMessage](#), [ImapGetMessageHeaders](#), [ImapGetMessageId](#), [ImapGetMessageSender](#)

ImapGetMessageSender Function

```
INT WINAPI ImapGetMessageSender(  
    HCLIENT hClient,  
    UINT nMessageId,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

The **ImapGetMessageSender** function returns the sender's address for the specified message.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve header value from. This value must be greater than zero. The first message in the mailbox is message number one.

lpszAddress

Pointer to a string buffer that will contain the address of the message sender.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, it returns the length of the address. If the sender cannot be determined, the function will return a value of zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMessageSender** function returns the email address specified in the Return-Path header field. This allows an application to be able to easily determine the sender, without parsing the header or downloading the contents of the message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapGetHeaderValue](#), [ImapGetMessageHeaders](#), [ImapGetMessageId](#)

ImapGetMessageSize Function

```
DWORD WINAPI ImapGetMessageSize(  
    HCLIENT hClient,  
    UINT nMessageId  
);
```

The **ImapGetMessageSize** function returns the size of the specified message.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve size of. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the function succeeds, the return value is the size of the specified message in bytes. If the function fails, the return value is IMAP_ERROR. To get extended error information, call

ImapGetLastError.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

See Also

[ImapGetHeaderValue](#), [ImapGetMessageHeaders](#), [ImapGetMessageId](#), [ImapGetMessageSender](#)

ImapGetMessageUid Function

```
DWORD WINAPI ImapGetMessageUid(  
    HCLIENT hClient,  
    UINT nMessageId  
);
```

The **ImapGetMessageUid** function returns the unique identifier (UID) for the specified message in the current mailbox.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to obtain the unique identifier for. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the function succeeds, it returns a non-zero value. If no unique identifier is assigned to the message, the function will return zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetMessageUid** function returns an unsigned integer value which specifies a unique identifier for this message. The actual value is determined by the server and should be considered opaque.

An application can use the message UID value in combination with the mailbox UID in order to uniquely identify a message on the server. However, the application must take into consideration that the IMAP server can reassign new message UIDs when the mailbox is modified. If the mailbox and message UIDs are being stored on the local system to track what messages have been retrieved from the server, the application must check the UID of the mailbox whenever it is selected. If the mailbox UID has changed, this means that the UIDs for the messages in that mailbox may have changed. The client should resynchronize with the server, and update it's local copy of that mailbox.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

See Also

[ImapGetCurrentMailbox](#), [ImapGetMailboxStatus](#), [ImapGetMailboxUID](#)

ImapGetNewMessages Function

```
INT WINAPI ImapGetNewMessages(  
    HCLIENT hClient,  
    UINT* lpnMessageIds,  
    INT nMaxMessages  
);
```

The **ImapGetNewMessages** function returns the message identifiers for those messages that have recently been added to the mailbox and have not been read.

Parameters

hClient

Handle to the client session.

lpnMessageIds

A pointer to an array of unsigned integers that will contain the message identifiers of those messages that have recently been added to the mailbox and have not been read. This parameter may be NULL, in which case the function will return the number of new messages but will not return their identifiers.

nMaxMessages

The maximum number of message identifiers that may be stored in the *lpnMessageIds* array. If the *lpnMessageIds* parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the number of new messages in the current mailbox. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The message identifiers returned by this function are only valid until the mailbox is expunged or another mailbox is selected. Once a message has been read using **ImapGetMessage** or **ImapStoreMessage**, it is no longer considered to be a new message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

See Also

[ImapGetDeletedMessages](#), [ImapGetMessageCount](#), [ImapGetMessageFlags](#),
[ImapGetUnseenMessages](#), [ImapSearchMailbox](#)

ImapGetNextMailbox Function

```
INT WINAPI ImapGetNextMailbox(  
    HCLIENT hClient,  
    LPTSTR lpszMailbox,  
    INT nMaxLength  
    LPDWORD lpdwFlags  
);
```

The **ImapGetNextMailbox** function returns the name of the next matching mailbox.

Parameters

hClient

Handle to the client session.

lpszMailbox

A pointer to a string buffer which will contain the next matching mailbox. This parameter cannot be NULL. A minimum buffer size of at least 128 character is recommended.

nMaxLength

Specifies the maximum length of the string buffer. The maximum length of the buffer should be large enough to accommodate most path names on the IMAP server.

lpdwFlags

A pointer to an unsigned integer which will contain the mailbox flags for the next matching mailbox. This parameter may be NULL, in which case the mailbox flags are not returned. Otherwise, one or more of the following bit flags may be returned:

Constant	Description
IMAP_FLAG_NOINFERIORS	The mailbox does not contain any sub-mailboxes. In the IMAP protocol, these are referred to as inferior hierarchical mailbox names.
IMAP_FLAG_NOSELECT	The mailbox cannot be selected or examined. This flag is typically used by servers to indicate that the mailbox name refers to a directory on the server, not a mailbox file.
IMAP_FLAG_MARKED	The mailbox is marked as being of interest to a client. If this flag is used, it typically means that the mailbox contains messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.
IMAP_FLAG_UNMARKED	The mailbox is marked as not being of interest to a client. If this flag is used, it typically means that the mailbox does not contain any messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.

Return Value

If the function succeeds, it returns the length of the mailbox name. If an error occurs, the function

returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetNextMailbox** function returns the next matching mailbox name. When the last mailbox has been returned, the next call to this function will result in an error, with the last error code set to ST_ERROR_NO_MORE_MAILBOXES.

Subscribed mailboxes are those which were specified using the **ImapSubscribeMailbox** function. Marked mailboxes are typically those which have some special importance to the user.

For more information about enumerating the available mailboxes on the IMAP server, refer to the **ImapGetFirstMailbox** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapDeleteMailbox](#), [ImapGetFirstMailbox](#), [ImapGetMailboxStatus](#), [ImapRenameMailbox](#), [ImapSelectMailbox](#)

ImapGetResultCode Function

```
INT WINAPI ImapGetResultCode(  
    HCLIENT hClient  
);
```

The **ImapGetResultCode** function reads the result code returned by the server in response to a command. The result code is an integer value, and indicates if the operation succeeded or failed.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the result code. If the function fails, it returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The following result codes may be returned by the IMAP server:

Constant	Description
IMAP_RESULT_UNKNOWN	An unknown result code was returned by the server.
IMAP_RESULT_OK	The previous command completed successfully. The result string contains information about the results of the command.
IMAP_RESULT_NO	The previous command could not be completed. The result string contains information about why the command failed.
IMAP_RESULT_BAD	The previous command could not be completed, the command may be invalid or not supported on the server. The result string contains information about why the command failed.
IMAP_RESULT_CONTINUE	The command has executed and is waiting for additional data from the client.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[ImapCommand](#), [ImapGetResultString](#)

ImapGetResultString Function

```
INT WINAPI ImapGetResultString(  
    HCLIENT hClient,  
    LPTSTR lpszResult,  
    INT nMaxLength  
);
```

The **ImapGetResultString** function returns the last message sent by the server along with the result code.

Parameters

hClient

Handle to the client session.

lpszResult

A pointer to the buffer that will contain the result string returned by the server.

nMaxLength

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetResultString** function is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **ImapGetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapCommand](#), [ImapGetResultCode](#)

ImapGetSecurityInformation Function

```
BOOL WINAPI ImapGetSecurityInformation(  
    HCLIENT hClient,  
    LPSECURITYINFO lpSecurityInfo  
);
```

The **ImapGetSecurityInformation** function returns security protocol, encryption and certificate information about the current client connection.

Parameters

hClient

Handle to the client session.

lpSecurityInfo

A pointer to a [SECURITYINFO](#) structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **ImapGetLastError**.

Remarks

This function is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value `SECURITY_PROTOCOL_NONE`.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;  
  
securityInfo.dwSize = sizeof(SECURITYINFO);  
if (ImapGetSecurityInformation(hClient, &securityInfo))  
{  
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)  
    {  
        MessageBox(NULL, _T("The connection is not secure"),  
                    _T("Connection"), MB_OK);  
    }  
    else  
    {  
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)  
        {  
            MessageBox(NULL, _T("The connection is secure"),  
                        _T("Connection"), MB_OK);  
        }  
        else  
        {  
            MessageBox(NULL, _T("The server certificate not valid"),  
                        _T("Connection"), MB_OK);  
        }  
    }  
}
```

```
}  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapAsyncConnect](#), [ImapConnect](#), [ImapCreateSecurityCredentials](#), [ImapDeleteSecurityCredentials](#)

ImapGetStatus Function

```
INT WINAPI ImapGetStatus(  
    HCLIENT hClient  
);
```

The **ImapGetStatus** function returns the current status of the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetStatus** function returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Constant	Description
1	IMAP_STATUS_IDLE	The client is current idle and not sending or receiving data.
2	IMAP_STATUS_CONNECT	The client is establishing a connection with the server.
3	IMAP_STATUS_READ	The client is reading data from the server.
4	IMAP_STATUS_WRITE	The client is writing data to the server.
5	IMAP_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

See Also

[ImapIsBlocking](#), [ImapIsConnected](#), [ImapIsReadable](#), [ImapIsWritable](#)

ImapGetTimeout Function

```
INT WINAPI ImapGetTimeout(  
    HCLIENT hClient  
);
```

The **ImapGetTimeout** function returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmav10.lib

See Also

[ImapConnect](#), [ImapIsReadable](#), [ImapIsWritable](#), [ImapRead](#), [ImapSetTimeout](#), [ImapWrite](#)

ImapGetTransferStatus Function

```
INT WINAPI ImapGetTransferStatus(  
    HCLIENT hClient,  
    LPIMAPTRANSFERSTATUS lpStatus  
);
```

The **ImapGetTransferStatus** function returns information about the current file transfer in progress.

Parameters

hClient

Handle to the client session.

lpStatus

A pointer to an [IMAPTRANSFERSTATUS](#) structure which contains information about the status of the current file transfer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `IMAP_ERROR`. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapGetTransferStatus** function returns information about the current file transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no file currently being transferred, this function will return the status of the last successful transfer made by the client.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmapi10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapEnableEvents](#), [ImapGetStatus](#), [ImapRegisterEvent](#)

ImapGetUnseenMessages Function

```
INT WINAPI ImapGetUnseenMessages(  
    HCLIENT hClient,  
    UINT * lpnMessageIds,  
    INT nMaxMessages  
);
```

The **ImapGetUnseenMessages** function returns the message identifiers for those messages that have not been read.

Parameters

hClient

Handle to the client session.

lpnMessageIds

A pointer to an array of unsigned integers that will contain the message identifiers of those messages that have not been read. This parameter may be NULL, in which case the function will return the number of unseen messages but will not return their identifiers.

nMaxMessages

The maximum number of message identifiers that may be stored in the *lpnMessageIds* array. If the *lpnMessageIds* parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the number of unseen messages in the current mailbox. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The message identifiers returned by this function are only valid until the mailbox is expunged or another mailbox is selected. Once a message has been read using **ImapGetMessage** or **ImapStoreMessage**, it is no longer considered to be an unseen message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

See Also

[ImapGetDeletedMessages](#), [ImapGetMessageCount](#), [ImapGetMessageFlags](#),
[ImapGetNewMessages](#), [ImapSearchMailbox](#)

ImapIdle Function

```
INT WINAPI ImapIdle(  
    HCLIENT hClient,  
    UINT nTimeout,  
    DWORD dwOptions,  
    IMAPIDLEPROC lpfnIdleProc,  
    DWORD_PTR dwParam  
);
```

The **ImapIdle** function enables mailbox status monitoring for the client session, allowing the client to receive notifications from the server whenever a new message arrives or a message is expunged from the currently selected mailbox. This is typically used as an alternative to the client periodically polling the server for status information.

Parameters

hClient

Handle to the client session.

nTimeout

Specifies the timeout period in seconds to wait for a notification from the server. This parameter is only used when the IMAP_IDLE_WAIT option has been specified.

dwOptions

Specifies the options which should be used when enabling idle monitoring. The following options are supported:

Constant	Description
IMAP_IDLE_NOWAIT	The function should return immediately after idle processing has been enabled. When this option is used, the application may continue to perform other functions while the client session is monitored for status updates sent by the server. The client will continue to monitor status changes until an IMAP command issued or the client disconnects from the server.
IMAP_IDLE_WAIT	The function should wait until the server sends a status update, or until the timeout period is reached. The client will stop monitoring status changes when the function returns. If this option is used in a single-threaded application, normal message processing can be impeded, causing the application to appear non-responsive until the timeout period is reached. It is strongly recommended that single-threaded applications with a user interface specify the IMAP_IDLE_NOWAIT option instead.

lpfnIdleProc

A pointer to an [IMAPIDLEPROC](#) callback function that will be invoked whenever the server sends an update notification to the client. This parameter must specify a valid function address and cannot be a NULL pointer.

dwParam

A user-defined value that is passed back to the caller whenever the callback function is invoked. This can be used to provide additional state information to the client. If it is not needed, the

caller should use a value of zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `IMAP_ERROR`. To get extended error information, call **ImapGetLastError**.

Remarks

Many IMAP servers support the ability to asynchronously send status updates to the client, rather than have the client periodically poll the server. The client enables this feature by calling **ImapIdle** and providing the address of a callback function that will be invoked whenever the server sends an update notification to the client. Typically these updates inform the client that a new message has arrived or that a message has been expunged from the mailbox.

The **ImapIdle** function can operate in two modes, based on the options specified by the caller. If the option `IMAP_IDLE_NOWAIT` is specified, the function begins monitoring the client session and returns control immediately to the caller. If the server sends a update notification, the callback function will be invoked with information about the status change. If the option `IMAP_IDLE_WAIT` is specified, the function will block waiting for the server to send a notification message to the client. The function will return when either a message is received or the timeout period is exceeded.

Sending an IMAP command to the server will cause the client to stop monitoring the session for status changes. To explicitly stop monitoring the session, use the **ImapCancel** function. To determine if the current client session is being monitored, use the **ImapGetIdleThreadId** function. A non-zero return value indicates that the client session is idle and being monitored.

This function works by sending the `IDLE` command to the server and starting a worker thread which monitors the connection and looks for untagged responses issued by the server. Callbacks will be invoked for `EXISTS`, `EXPUNGE` and `RECENT` messages. Note that some servers may periodically send untagged `OK` messages to the client, indicating that the connection is still active; these messages are explicitly ignored. Because the monitoring is performed in a different thread, the callback function is invoked in the context of that thread. Client event notifications are disabled while inside the callback function, and the **ImapIdle** function cannot be used to restart monitoring from within a callback function.

Applications should not perform any operation that takes a significant amount of time or updates the user interface from within the callback function. Instead, use flags or send application defined messages to indicate a change in state. For example, if the server sends a notification that a new email message has arrived, the application should not attempt to read the new message and update the user interface from within the callback function. Instead, it could use the **PostMessage** function to send an application-defined message to the UI thread indicating the change in state. The application would have a message handler for that Windows message and update the user interface, indicating that a new message has arrived.

An application should never make an assumption about how a particular server may send update notifications to the client. Servers can be configured to use different intervals at which notifications are sent. For example, a server may send new message notifications immediately, but may periodically notify the client when a message has been expunged. Alternatively, a server may only send notifications at fixed intervals, in which case the client would not be notified of any new messages until the interval period is reached. It is not possible for a client to know what a particular server's update interval is. Applications that require that degree of control should not use **ImapIdle** and should poll the server instead.

Example

```

// Begin monitoring the client session for status changes; when a new
// message arrives or a message is expunged, the UpdateHandler callback
// function will be invoked.
BOOL MonitorUpdates(HWND hwndNotify)
{
    INT nResult;

    nResult = ImapIdle(hClient, 0, IMAP_IDLE_NOWAIT, UpdateHandler, hwndNotify);

    if (nResult == IMAP_ERROR)
    {
        ShowError(ImapGetLastError());
        return FALSE;
    }

    return TRUE;
}

// This function is called whenever the server notifies the client
// that a new message has arrived or a message has been expunged from
// the current mailbox
#define WM_APP_NEWMESSAGE (WM_APP+1)
#define WM_APP_EXPUNGED (WM_APP+2)

VOID CALLBACK UpdateHandler(HCLIENT hClient, UINT nUpdateId, UINT nMessageId,
DWORD_PTR dwParam)
{
    switch (nUpdateId)
    {
    case IMAP_UPDATE_MESSAGE:
        {
            // Send a message indicating that a new message has arrived
            PostMessage((HWND)dwParam, WM_APP_NEWMESSAGE, nMessageId, 0);
        }
        break;

    case IMAP_UPDATE_EXPUNGE:
        {
            // Send a message indicating that a message has been expunged
            PostMessage((HWND)dwParam, WM_APP_EXPUNGED, nMessageId, 0);
        }
        break;
    }
}

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapGetIdleThreadId](#), [ImapIdleProc](#)

ImapIdleProc Function

```
VOID CALLBACK ImapIdleProc(  
    HCLIENT hClient,  
    UINT nUpdateId,  
    UINT nMessageId,  
    DWORD_PTR dwParam  
);
```

The **ImapIdleProc** function is an application-defined callback function that is invoked whenever the server sends a status update to the client. For more information about status monitoring, refer to the **ImapIdle** function.

Parameters

hClient

Handle to the client session.

nUpdateId

An unsigned integer which specifies the type of update notification that has been sent by the server. It may be one of the following values:

Constant	Description
IMAP_UPDATE_UNKNOWN	The server has sent an unrecognized notification message. The value of the <i>nMessageId</i> argument is undefined for this type of notification. This does not necessarily reflect an error condition, as some servers may send additional notification messages beyond the standard EXISTS, EXPUNGE and RECENT messages. Most applications should ignore this type of notification.
IMAP_UPDATE_MESSAGE	The server has sent notification message to the client indicating that a new message has arrived. The <i>nMessageId</i> argument will contain the message number for the new message. Typically this update notification occurs shortly after the new message has been stored in the current mailbox.
IMAP_UPDATE_EXPUNGE	The server has sent a notification message to the client indicating that a message has been removed from the current mailbox. The <i>nMessageId</i> argument will contain the message number for the message that has been removed. It is recommended that the application re-examine the mailbox when this notification is received. Typically this notification is only sent periodically by the server, and may not be sent immediately after a message has been expunged from the mailbox.
IMAP_UPDATE_MAILBOX	The server has sent notification message to the client indicating that the state of the mailbox has changed. The <i>nMessageId</i> argument is not used with this notification. This message is sent periodically by the server and may not be sent immediately after a new message arrives or a

message is flagged as unread. It is recommended that the application re-examine the mailbox when this notification is received.

nMessageId

An unsigned integer which specifies the message number associated with the status change. Note that this argument is not used with the IMAP_UPDATE_MAILBOX notification and will contain a value of zero.

dwParam

A user-defined integer value which was specified when the **ImapIdle** function was called.

Return Value

None.

Remarks

An application must enable this callback function by passing its address to the **ImapIdle** function. The **ImapIdleProc** function is a placeholder for the application-defined function name.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmav10.lib

See Also

[ImapGetIdleThreadId](#), [ImapIdle](#)

ImapInitialize Function

```
BOOL WINAPI ImapInitialize(  
    LPCTSTR lpszLicenseKey,  
    LPINITDATA lpData  
);
```

The **ImapInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpData

Pointer to an INITDATA data structure. This parameter may be NULL if the initialization data for the library is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **ImapGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must insure that this function is called with a valid license key.

If the *lpData* argument is specified, it must point to an INITDATA structure which has been initialized by setting the *dwSize* member to the size of the structure. All other structure members should be set to zero. If the function is successful, then the INITDATA structure will be filled with identifying information about the library.

Although it is only required that **ImapInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **ImapUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **ImapInitialize** or **ImapUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

ImapIsBlocking Function

```
BOOL WINAPI ImapIsBlocking(  
    HCLIENT hClient  
);
```

The **ImapIsBlocking** function is used to determine if the client is currently performing a blocking operation.

Parameters

hClient

Handle to the client session.

Return Value

If the client is performing a blocking operation, the function returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the function returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking function may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **ImapIsBlocking** function can be used to determine if the client is already blocked, and if so, take some other action such as warning the user that they must wait for the operation to complete.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

See Also

[ImapCancel](#), [ImapIsConnected](#), [ImapIsReadable](#), [ImapIsWritable](#)

ImapIsConnected Function

```
BOOL WINAPI ImapIsConnected(  
    HCLIENT hClient  
);
```

The **ImapIsConnected** function is used to determine if the client is currently connected to a server.

Parameters

hClient

Handle to the client session.

Return Value

If the client is connected to a server, the function returns a non-zero value. If the client is not connected, or the client handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapIsBlocking](#), [ImapIsReadable](#), [ImapIsWritable](#)

ImapIsReadable Function

```
BOOL WINAPI ImapIsReadable(  
    HCLIENT hClient,  
    INT nTimeout,  
    LPDWORD lpdwAvail  
);
```

The **ImapIsReadable** function is used to determine if data is available to be read from the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the function returns a non-zero value. If the client cannot read any data, the function returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

See Also

[ImapGetStatus](#), [ImapIsBlocking](#), [ImapIsConnected](#), [ImapIsWritable](#), [ImapRead](#)

ImapIsWritable Function

```
BOOL WINAPI ImapIsWritable(  
    HCLIENT hClient,  
    INT nTimeout  
);
```

The **ImapIsWritable** function is used to determine if data can be written to the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the function returns a non-zero value. If the client cannot write any data, the function returns zero.

Remarks

Although this function can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

See Also

[ImapGetStatus](#), [ImapIsBlocking](#), [ImapIsConnected](#), [ImapIsReadable](#), [ImapWrite](#)

ImapLogin Function

```
INT WINAPI ImapLogin(  
    HCLIENT hClient,  
    UINT nAuthType,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword  
);
```

The **ImapLogin** function authenticates the specified user in on the server. This function must be called after the connection has been established, and before attempting to retrieve messages or perform any other function on the server.

Parameters

hClient

Handle to the client session.

nAuthType

Identifies the type of authentication that should be used when the client logs in to the mail server. The following authentication methods are supported:

Constant	Description
IMAP_AUTH_LOGIN	Standard cleartext username and password is sent to the server. This authentication method is supported by all servers. Note that some servers may only support LOGIN authentication if a secure connection has been established.
IMAP_AUTH_PLAIN	Login using the PLAIN authentication mechanism as defined in RFC 4959. This authentication method is supported by most servers, although some may require that client establish a secure connection.
IMAP_AUTH_XOAUTH2	This authentication type will use the XOAUTH2 method to authenticate the client session. This authentication method does not require the user password, instead the <i>lpszPassword</i> parameter must specify the OAuth 2.0 bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this function will fail.
IMAP_AUTH_BEARER	This authentication type will use the OAUTHBEARER method to authenticate the client session as defined in RFC 7628. This authentication method does not require the user password, instead the <i>lpszPassword</i> parameter must specify the OAuth 2.0 bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this function will fail.
IMAP_AUTH_ANONYMOUS	Login using the anonymous Simple Authentication and Security Layer (SASL) mechanism as defined in RFC 4505. If this authentication method is specified, the

<p><i>lpszUserName</i> parameter should specify a name or email address acceptable to the mail server. The <i>lpszPassword</i> parameter is ignored and may be NULL.</p>
--

lpszUserName

A null terminated string which specifies the user name to be used to authenticate the current client session.

lpszPassword

A null terminated string which specifies the password to be used when authenticating the current client session. If you are using the IMAP_AUTH_XOAUTH2 or IMAP_AUTH_BEARER authentication methods, this parameter is not a password, instead it specifies the OAuth 2.0 bearer token provided by the mail service.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

In some cases, the user may be pre-authenticated by the server. In this case, the function will fail with the last error set to ST_ERROR_ALREADY_AUTHENTICATED. If a particular authentication method is not supported by the server, the last error will be set to ST_ERROR_INVALID_AUTHENTICATION_TYPE. For compatibility with the greatest number of servers, it is recommended that you use IMAP_AUTH_LOGIN as the authentication method.

You should only use an OAuth 2.0 authentication method if you understand the process of how to request the access token. Obtaining an access token requires registering your application with the mail service provider (e.g.: Microsoft or Google), getting a unique client ID associated with your application and then requesting the access token using the appropriate scope for the service. Obtaining the initial token will typically involve interactive confirmation on the part of the user, requiring they grant permission to your application to access their mail account.

The IMAP_AUTH_XOAUTH2 and IMAP_AUTH_BEARER authentication methods are similar, but they are not interchangeable. Both use an OAuth 2.0 bearer token to authenticate the client session, but they differ in how the token is presented to the server. It is currently preferable to use the XOAUTH2 method because it is more widely available and some service providers do not yet support the OAUTHBEARER method.

Your application should not store an OAuth 2.0 bearer token for later use. They have a relatively short lifespan, typically about an hour, and are designed to be used with that session. You should specify offline access as part of the OAuth 2.0 scope if necessary and store the refresh token provided by the service. The refresh token has a much longer validity period and can be used to obtain a new bearer token when needed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapAsyncConnect](#), [ImapConnect](#), [ImapDisconnect](#), [ImapInitialize](#), [ImapUninitialize](#)

ImapOpenMessage Function

```
INT WINAPI ImapOpenMessage(  
    HCLIENT hClient,  
    UINT nMessageId,  
    DWORD dwReserved  
);
```

The **ImapOpenMessage** function opens the specified message on the server.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve. This value must be greater than zero. The first message in the mailbox is message number one.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapOpenMessage** function uses the FETCH command to access the specified message on the server. The client can then use the the **ImapRead** function to read the contents of the message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapCloseMessage](#), [ImapGetMessage](#), [ImapGetMessageHeaders](#), [ImapOpenMessageEx](#), [ImapRead](#), [ImapStoreMessage](#)

ImapOpenMessageEx Function

```
INT WINAPI ImapOpenMessageEx(  
    HCLIENT hClient,  
    UINT nMessageId,  
    UINT nMessagePart,  
    DWORD dwOffset,  
    LPDWORD lpdwLength,  
    DWORD dwOptions  
);
```

The **ImapOpenMessageEx** function opens a message or a specific part of a multipart message in the current mailbox. The message data may also be limited a specific byte offset and length, which can be useful for previewing the contents.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve. This value must be greater than zero. The first message in the mailbox is message number one.

nMessagePart

The message part that will be retrieved. A value of zero specifies that the complete message should be returned. If the message is a multipart MIME message, message parts start with a value of one.

dwOffset

The byte offset into the message. This parameter can be used in conjunction with the *lpdwLength* parameter to return a specific part of a message. A value of zero specifies the beginning of the message.

lpdwLength

A pointer to an unsigned integer value which should be initialized to the maximum number of bytes to be read, and will contain the size of the message when the function returns. To specify the entire message, from the offset specified by the *dwOffset* parameter to the end of the message, initialize the *lpdwLength* parameter to a value of -1. This parameter may be NULL if the message size is not needed.

dwOptions

The low order word of this parameter specifies how the message data will be returned. It may be one of the following values:

Constant	Description
IMAP_SECTION_DEFAULT	All headers and the complete body of the specified message or message part are to be retrieved. The client application is responsible for parsing the header block. If the message is a MIME multipart message and the complete message is returned, the application is responsible for parsing the individual message parts if necessary.

IMAP_SECTION_HEADER	All headers for the specified message or message part are to be retrieved. The client application is responsible for parsing the header block.
IMAP_SECTION_MIMEHEADER	The MIME headers for the specified message or message are to be retrieved. Only those header fields which are used in MIME messages, such as Content-Type will be returned to the client. This is typically useful when processing the header for a multipart message which contains file attachments. The client application is responsible for parsing the header block.
IMAP_SECTION_BODY	The body of the specified message or message part is to be retrieved. For a MIME formatted message, this may include data that is uuencoded or base64 encoded. The application is responsible for decoding this data.
IMAP_SECTION_PREVIEW	The message header or body is being previewed and should not be marked as read by the server. This prevents the message from having the IMAP_FLAG_SEEN flag from being automatically set when the message data is retrieved.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapOpenMessageEx** function uses the FETCH command to access the specified message on the server. The client can then use the **ImapRead** function to read the contents of the message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmavp10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapCloseMessage](#), [ImapGetMessage](#), [ImapGetMessageHeaders](#), [ImapOpenMessage](#), [ImapRead](#), [ImapStoreMessage](#)

ImapRead Function

```
INT WINAPI ImapRead(  
    HCLIENT hClient,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **ImapRead** function reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hClient

Handle to the client session.

lpBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that there is no more data available to be read. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

When **ImapRead** is called and the client is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

See Also

[ImapGetMessage](#), [ImapGetMessageHeaders](#), [ImapIsBlocking](#)

ImapRegisterEvent Function

```
INT WINAPI ImapRegisterEvent(  
    HCLIENT hClient,  
    UINT nEvent,  
    IMAPEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

The **ImapRegisterEvent** function registers a callback function for the specified event.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Constant	Description
IMAP_EVENT_CONNECT	The connection to the server has completed.
IMAP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
IMAP_EVENT_READ	Data is available to read by the client. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the calling process is in asynchronous mode.
IMAP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
IMAP_EVENT_TIMEOUT	The client has timed out while waiting for a response from the server. Note that under some circumstances this event can be generated for a non-blocking connection, such as when the client is establishing a secure connection.
IMAP_EVENT_CANCEL	The client has canceled the current operation.
IMAP_EVENT_COMMAND	The client has processed a command that was sent to the server. The result code and result string can be used to determine if the response to the command.
IMAP_EVENT_PROGRESS	This event notification is sent periodically during lengthy blocking operations, such as retrieving a complete message from the server.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **ImapEventProc** callback

function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapRegisterEvent** function associates a callback function with a specific event. The event handler is an **ImapEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This function is typically used to register an event handler that is invoked while a message is being retrieved. The IMAP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one IMAP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

See Also

[ImapDisableEvents](#), [ImapEnableEvents](#), [ImapEventProc](#), [ImapFreezeEvents](#)

ImapRenameMailbox Function

```
INT WINAPI ImapRenameMailbox(  
    HCLIENT hClient,  
    LPCTSTR lpszOldMailbox,  
    LPCTSTR lpszNewMailbox  
);
```

The **ImapRenameMailbox** function renames an existing mailbox.

Parameters

hClient

Handle to the client session.

lpszOldMailbox

A pointer to a string which specifies the mailbox to be renamed.

lpszNewMailbox

A pointer to a string which specifies the new mailbox name.

Return Value

If the function succeeds, it returns zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapRenameMailbox** function renames an existing mailbox on the server. The new mailbox name cannot exist on the server, or the function will fail.

If the existing mailbox name contains inferior hierarchical names (mailboxes under the specified mailbox) then those mailboxes will also be renamed. For example, if the mailbox "Mail/Pictures" contains two mailboxes, "Personal" and "Work" and it is renamed to "Mail/Images" then the two mailboxes under it would be automatically renamed to "Mail/Images/Personal" and "Mail/Images/Work".

If the mailbox being renamed is the currently selected mailbox, the current mailbox will be unselected and any messages marked for deletion will be expunged. The new mailbox name will then automatically be re-selected. To prevent deleted messages from being removed from the mailbox prior to being renamed, use the **ImapUnselectMailbox** function to unselect the current mailbox before calling **ImapRenameMailbox**. Note that if the rename operation fails, the client may be left in an unselected state.

It is permitted to rename the special mailbox INBOX. In this case, the messages will be moved from the INBOX mailbox to the new mailbox. If the INBOX mailbox is currently selected, the new mailbox will not automatically be selected. INBOX will remain the selected mailbox.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapCreateMailbox](#), [ImapDeleteMailbox](#), [ImapGetFirstMailbox](#), [ImapGetNextMailbox](#)

ImapReselectMailbox Function

```
BOOL WINAPI ImapReselectMailbox(  
    HCLIENT hClient,  
    LPIMAPMAILBOX lpMailboxInfo  
);
```

The **ImapReselectMailbox** function reselects the current mailbox and returns updated information about the status of the mailbox.

Parameters

hClient

Handle to the client session.

lpMailboxInfo

A pointer to an [IMAPMAILBOX](#) structure which contains updated information about the mailbox when the function returns. This parameter may be NULL if the caller does not require any information about the mailbox.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `IMAP_ERROR`. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapReselectMailbox** function forces the current mailbox to be reselected and returns updated information about the status of the mailbox. Deleted messages are not expunged from the mailbox and remain marked for deletion.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmapi10.lib`

See Also

[ImapExamineMailbox](#), [ImapSelectMailbox](#), [ImapUnselectMailbox](#)

ImapSearchMailbox Function

```
LONG WINAPI ImapSearchMailbox(  
    HCLIENT hClient,  
    LPCTSTR lpszCriteria,  
    LPCTSTR lpszCharacterSet,  
    DWORD dwReserved,  
    UINT* lpnMessageIds,  
    LONG nMaxMessages  
);
```

The **ImapSearchMailbox** function searches the current mailbox for messages that match the specified criteria, returning matching message identifiers.

Parameters

hClient

Handle to the client session.

lpszCriteria

A pointer to a string which specifies the search criteria.

lpszCharacterSet

A pointer to a string which specifies the character set to use when searching the mailbox. If this parameter is NULL or an empty string, the default UTF-8 character set will be used.

dwReserved

A reserved parameter which should be set to the value 0.

lpnMessageIds

A pointer to an array of unsigned integers that will contain the message identifiers of those messages which match the search criteria in the current mailbox. This parameter may be NULL, in which case the function will return the number of matching messages but will not return their identifiers.

nMaxMessages

The maximum number of message identifiers that may be stored in the *lpnMessageIds* array. If the *lpnMessageIds* parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is the number of messages that meet the search criteria in the current mailbox. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapSearchMailbox** function is used to search a mailbox for messages which match a given criteria and return a list of the matching message identifiers. The search criteria is composed of one or more search keywords and an optional value to match against. String searches are not case sensitive and partial matches in the message are returned.

The following search keywords are recognized:

Keyword	Description
ANSWERED	Match those messages which have the IMAP_FLAG_ANSWERED flag set.

BCC <i>address</i>	Match those messages which contain the specified address in the BCC header field.
BEFORE <i>date</i>	Match those messages which were added to the mailbox prior to the specified date.
BODY <i>string</i>	Match those messages where the body contains the specified string.
CC <i>address</i>	Match those messages which contain the specified address in the CC header field.
DELETED	Match those messages which have the IMAP_FLAG_DELETED flag set.
DRAFT	Match those messages which have the IMAP_FLAG_DRAFT flag set.
FLAGGED	Match those messages which have the IMAP_FLAG_URGENT flag set.
FROM <i>address</i>	Match those messages which contain the specified address in the FROM header field.
HEADER <i>field string</i>	Match those messages which contain the string in the specified header field. If no string is specified, then all messages which contain the header will be matched.
LARGER <i>size</i>	Match those messages which are larger than the specified size in bytes.
NEW	Match those messages which have the IMAP_FLAG_RECENT flag set, but not the IMAP_FLAG_SEEN flag.
OLD	Match those messages which do not have the IMAP_FLAG_RECENT flag set.
ON <i>date</i>	Match those messages which were added on the specified date.
RECENT	Match those messages which have the IMAP_FLAG_RECENT flag set.
SEEN	Match those messages which have the IMAP_FLAG_SEEN flag set.
SENTBEFORE <i>date</i>	Match those messages whose Date header value is earlier than the specified date.
SENTON <i>date</i>	Match those messages whose Date header value is the same as the specified date.
SENTSINCE <i>date</i>	Match those messages whose Date header value is later than the specified date.
SINCE <i>date</i>	Match those messages added to the mailbox after the specified date.
SMALLER <i>size</i>	Match those messages which are smaller than the specified size in bytes.
SUBJECT <i>string</i>	Match those messages whose Subject header contains the specified string.
TEXT <i>string</i>	Match those messages whose headers or body contains the specified string.
TO <i>address</i>	Match those messages which contain the specified address in the TO header field.
UID <i>sequence</i>	Match those messages with unique identifiers in the sequence set.

UNANSWERED	Match those messages which do not have the IMAP_FLAG_ANSWERED flag set.
UNDELETED	Match those messages which do not have the IMAP_FLAG_DELETED flag set.
UNDRAFT	Match those messages which do not have the IMAP_FLAG_DRAFT flag set.
UNFLAGGED	Match those messages which do not have the IMAP_FLAG_URGENT flag set.
UNSEEN	Match those messages which do not have the IMAP_FLAG_SEEN flag set.

In addition to the listed keywords, the keyword NOT may prefix a keyword to return those messages which do not match the search criteria. For example, "NOT TO user@domain.com" would return those messages which were not addressed to user@domain.com.

If multiple search keywords are specified, the result is the intersection of all those messages which meet the search criteria. For example, a search criteria of "DELETED SINCE 1-Jan-2003" would return all those messages which are marked for deletion and were added to the mailbox after 1 January 2003.

Those search keywords which expect dates must be specified in format *dd-mmm-yyyy* where the month is the three letter abbreviation for the month name. Note that the internal date the message was added to the mailbox is not the same as the value of the Date header field in the message.

The UID keyword expects a one or more unique message identifiers. These values may provided as comma separated list, or a range delimited by a colon. For example, "UID 23000:24000" would return all those messages who have UIDs ranging from 23000 through to 24000.

The message identifiers returned by this function are only valid until the mailbox is expunged or another mailbox is selected.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapGetDeletedMessages](#), [ImapGetMessageCount](#), [ImapGetMessageFlags](#),
[ImapGetNewMessages](#), [ImapGetUnseenMessages](#)

ImapSelectMailbox Function

```
INT WINAPI ImapSelectMailbox(  
    HCLIENT hClient,  
    LPCTSTR lpszMailbox,  
    LPIMAPMAILBOX lpMailboxInfo  
);
```

The **ImapSelectMailbox** function selects the specified mailbox for read-write access.

Parameters

hClient

Handle to the client session.

lpszMailbox

A pointer to a string which specifies the new mailbox to be selected.

lpMailboxInfo

A pointer to an [IMAPMAILBOX](#) structure which contains information about the mailbox when the function returns. This parameter may be NULL if the caller does not require any information about the mailbox.

Return Value

If the function succeeds, it returns zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapSelectMailbox** function is used to select a mailbox in read-write mode. If the client has a different mailbox currently selected, that mailbox will be closed and any messages marked for deletion will be expunged. To prevent deleted messages from being removed from the previous mailbox, use the **ImapUnselectMailbox** function prior to selecting the new mailbox.

If an application wishes to update the information returned in the IMAPMAILBOX structure for the current mailbox, simply call **ImapSelectMailbox** again with the same mailbox name. Note that this will not cause any messages marked for deletion to be expunged.

The special case-insensitive mailbox name INBOX is used for new messages. Other mailbox names may or may not be case-sensitive depending on the IMAP server's operating system and implementation.

To access a mailbox in read-only mode, use the **ImapExamineMailbox** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapDeleteMailbox](#), [ImapExamineMailbox](#), [ImapGetFirstMailbox](#), [ImapGetMailboxStatus](#), [ImapGetNextMailbox](#), [ImapRenameMailbox](#), [ImapReselectMailbox](#), [ImapUnselectMailbox](#)

ImapSetLastError Function

```
VOID WINAPI ImapSetLastError(  
    DWORD dwErrorCode  
);
```

The **ImapSetLastError** function sets the last error code for the current thread.

Parameters

dwErrorCode

Specifies the last error code for the caller. A value of zero clears the last error code.

Return Value

None.

Remarks

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value error code such as FALSE, NULL, INVALID_CLIENT or IMAP_ERROR. Those functions which call **ImapSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **ImapGetLastError** function. The use of **ImapGetLastError** is optional. An application can call it to find out the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmapi10.lib

See Also

[ImapGetErrorString](#), [ImapGetLastError](#)

ImapSetMessageFlags Function

```
INT WINAPI ImapSetMessageFlags(  
    HCLIENT hClient,  
    UINT nMessageId,  
    UINT nMode  
    DWORD dwMessageFlags  
);
```

The **ImapSetMessageFlags** function returns the message flags for the specified message.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to obtain the message flags for. This value must be greater than zero. The first message in the mailbox is message number one.

nMode

An unsigned integer value which specifies one of the following modes which determines how the message flags are set:

Constant	Description
IMAP_FLAGS_REPLACE	All message flags are replaced with the flags specified by the <i>dwMessageFlags</i> parameter.
IMAP_FLAGS_ADD	The message flags specified by the <i>dwMessageFlags</i> parameter will be set for the message. Message flags that have been previously set will remain unmodified.
IMAP_FLAGS_REMOVE	The message flags specified by the <i>dwMessageFlags</i> parameter will be removed from the message. Message flags that are not specified will remain unmodified.

dwMessageFlags

An unsigned integer value which specifies one or more message flags. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
IMAP_FLAG_ANSWERED	The message has been answered.
IMAP_FLAG_DELETED	The message has been marked for deletion.
IMAP_FLAG_DRAFT	The message has not been completed and is marked as a draft copy.
IMAP_FLAG_URGENT	The message has been flagged for urgent or special attention.
IMAP_FLAG_RECENT	The message has been added to the mailbox recently.
IMAP_FLAG_SEEN	The message has been read.

Return Value

If the function succeeds, it returns IMAP_RESULT_OK. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[ImapDeleteMessage](#), [ImapGetMessageCount](#), [ImapGetMessageFlags](#)

ImapSetTimeout Function

```
INT WINAPI ImapSetTimeout(  
    HCLIENT hClient,  
    INT nTimeout  
);
```

The **ImapSetTimeout** function sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The timeout value is only used with blocking network operations.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmav10.lib

See Also

[ImapConnect](#), [ImapGetTimeout](#), [ImapRead](#)

ImapStoreMessage Function

```
INT WINAPI ImapStoreMessage(  
    HCLIENT hClient,  
    UINT nMessageId,  
    LPCTSTR lpszFileName  
);
```

The **ImapStoreMessage** function retrieves a message from the current mailbox and stores it in a local file or the system clipboard.

Parameters

hClient

Handle to the client session.

nMessageId

Number of the message to retrieve. This value must be greater than zero. The first message in the mailbox is message number one.

lpszFileName

Pointer to a string which specifies the file that the message will be stored in. If an empty string or NULL pointer is passed as an argument, the message is copied to the system clipboard.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapStoreMessage** function provides a method of retrieving and storing a message on the local system. The contents of the message is stored as a text file, using the specified file name. This function always causes the caller to block until the entire message has been retrieved, even if the client has been put in asynchronous mode.

If event handling is enabled, the IMAP_EVENT_PROGRESS event will fire periodically during the transfer of the message to the local system. An application can determine how much of the message has been retrieved by calling the **ImapGetTransferStatus** function.

To retrieve the message into a global memory buffer so that it can be passed to the MIME or SMTP libraries, use the **ImapGetMessage** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapGetMessage](#), [ImapGetMessageHeaders](#), [ImapGetTransferStatus](#)

ImapSubscribeMailbox Function

```
INT WINAPI ImapSubscribeMailbox(  
    HCLIENT hClient,  
    LPCTSTR lpszMailbox  
);
```

The **ImapSubscribeMailbox** function subscribes the user to the specified mailbox.

Parameters

hClient

Handle to the client session.

lpszMailbox

A pointer to a string which specifies the mailbox to subscribe to.

Return Value

If the function succeeds, it returns zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapSubscribeMailbox** function adds the specified mailbox to the current user's list of active or subscribed mailboxes. The user will remain subscribed to the mailbox across multiple sessions, until the **ImapUnsubscribeMailbox** function is called to remove the mailbox from the subscription list.

To list those mailboxes which the user has subscribed to, use the **ImapGetFirstMailbox** function and specify the IMAP_LIST_SUBSCRIBED option.

Note that if a user subscribes to a mailbox and that mailbox is later renamed or deleted, the mailbox will not be automatically removed from the user's subscription list. An application must not assume that because a mailbox name is included in the list of subscribed mailboxes, it exists and can be selected. To check if the mailbox exists, use the **ImapGetMailboxStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapExamineMailbox](#), [ImapGetFirstMailbox](#), [ImapGetNextMailbox](#), [ImapSelectMailbox](#), [ImapUnselectMailbox](#), [ImapUnsubscribeMailbox](#)

ImapUndeleteMessage Function

```
INT WINAPI ImapUndeleteMessage(  
    HCLIENT hClient,  
    UINT nMessageId  
);
```

The **ImapUndeleteMessage** function removes the deletion flag for the specified message.

Parameters

hClient

Handle to the client session.

nMessage

Number of message to undelete from the server. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmav10.lib

See Also

[ImapDeleteMessage](#), [ImapGetDeletedMessages](#), [ImapGetMessage](#), [ImapGetMessageCount](#), [ImapGetMessageFlags](#), [ImapReselectMailbox](#), [ImapUnselectMailbox](#)

ImapUninitialize Function

```
VOID WINAPI ImapUninitialize();
```

The **ImapUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **ImapInitialize** call before it can call any of the other library functions. When it has completed the use of library, the application must call **ImapUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There must be a call to **ImapUninitialize** for every successful call to **ImapInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

See Also

[ImapAsyncConnect](#), [ImapConnect](#), [ImapDisconnect](#), [ImapInitialize](#)

ImapUnselectMailbox Function

```
INT WINAPI ImapUnselectMailbox(  
    HCLIENT hClient,  
    BOOL bExpunge  
);
```

The **ImapUnselectMailbox** function unselects the current mailbox.

Parameters

hClient

Handle to the client session.

bExpunge

A boolean flag which determines if deleted messages will be expunged from the mailbox. A non-zero value specifies that messages that have been marked for deletion will be removed from the mailbox. A zero value specifies that no messages will be removed from the mailbox.

Return Value

If the function succeeds, it returns zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

See Also

[ImapDeleteMailbox](#), [ImapExamineMailbox](#), [ImapGetFirstMailbox](#), [ImapGetMailboxStatus](#), [ImapGetNextMailbox](#), [ImapRenameMailbox](#), [ImapReselectMailbox](#), [ImapSelectMailbox](#)

ImapUnsubscribeMailbox Function

```
INT WINAPI ImapUnsubscribeMailbox(  
    HCLIENT hClient,  
    LPCTSTR lpszMailbox  
);
```

The **ImapUnsubscribeMailbox** function unsubscribes the user from the specified mailbox.

Parameters

hClient

Handle to the client session.

lpszMailbox

A pointer to a string which specifies the mailbox to unsubscribe from.

Return Value

If the function succeeds, it returns zero. If an error occurs, the function returns IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The **ImapUnsubscribeMailbox** function removes the specified mailbox from the current user's list of active or subscribed mailboxes.

To list those mailboxes which the user has subscribed to, use the **ImapGetFirstMailbox** function and specify the IMAP_LIST_SUBSCRIBED option.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ImapExamineMailbox](#), [ImapGetFirstMailbox](#), [ImapGetNextMailbox](#), [ImapSelectMailbox](#), [ImapSubscribeMailbox](#), [ImapUnselectMailbox](#)

ImapWrite Function

```
INT WINAPI ImapWrite(  
    HCLIENT hClient,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **ImapWrite** function sends the specified number of bytes to the server.

Parameters

hClient

Handle to the client session.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is IMAP_ERROR. To get extended error information, call **ImapGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the IMAP_EVENT_WRITE asynchronous notification message before it resumes sending data.

The **ImapWrite** function should not be used to send commands to the IMAP server. Use the **ImapCommand** function instead.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmapi10.lib

See Also

[ImapCommand](#), [ImapIsBlocking](#), [ImapIsWritable](#), [ImapRead](#)

Internet Message Access Protocol Data Structures

- IMAPMAILBOX
- IMAPMAILBOXSTATUS
- IMAPMESSAGE
- IMAPTRANSFERSTATUS
- INITDATA
- SECURITYCREDENTIALS
- SECURITYINFO
- SYSTEMTIME

IMAPMAILBOX Structure

This structure contains information about a selected mailbox.

```
typedef struct _IMAPMAILBOX
{
    UINT    nMessages;
    UINT    nRecentMessages;
    UINT    nUnseenMessageId;
    DWORD   dwMailboxUID;
    DWORD   dwNextMessageUID;
    DWORD   dwFlags;
    DWORD   dwPermanentFlags;
    DWORD   dwAccessMode;
    DWORD   dwReserved1;
    DWORD   dwReserved2;
} IMAPMAILBOX, *LPIMAPMAILBOX;
```

Members

nMessages

A value specifies the total number of messages in the mailbox.

nRecentMessages

A value which specifies the number of new messages that have recently arrived in the mailbox.

nUnseenMessageId

A value which specifies the message ID of the first unseen message in the mailbox.

dwMailboxUID

A value which specifies a unique identifier for this mailbox which corresponds to the UIDVALIDITY value returned by the IMAP server. The actual value is determined by the server and should be considered opaque. The protocol specification requires that a mailbox's UID must not change unless the mailbox contents are modified or existing messages in the mailbox have been assigned new UIDs.

dwNextMessageUID

A value which specifies the predicted unique identifier that will be assigned to a new message in the mailbox. This corresponds to the UIDNEXT value returned by the IMAP server. The protocol specification requires that as long as the mailbox UID is unchanged, messages that are added to the mailbox will be assigned a UID greater than or equal to the next UID value.

dwFlags

A value which specifies one or more mailbox flags. One or more of the following values may be specified:

Constant	Description
IMAP_FLAG_NOINFERIORS	The mailbox does not contain any child mailboxes. In the IMAP protocol, these are referred to as inferior hierarchical mailbox names.
IMAP_FLAG_NOSELECT	The mailbox cannot be selected or examined. This flag is typically used by servers to indicate that the mailbox name refers to a directory on the server, not a mailbox file.
IMAP_FLAG_MARKED	The mailbox is marked as being of interest to a client. If

	this flag is used, it typically means that the mailbox contains messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.
IMAP_FLAG_UNMARKED	The mailbox is marked as not being of interest to a client. If this flag is used, it typically means that the mailbox does not contain any messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.

dwPermanentFlags

A value which specifies the message flags that a client can change permanently. If this value is zero, then no permanent flags are defined for the mailbox and the client may assume that all message flags may be set permanently. Otherwise, one or more of the following values may be specified:

Constant	Description
IMAP_FLAG_ANSWERED	The message has been answered.
IMAP_FLAG_DRAFT	The message has not been completed and is marked as a draft copy.
IMAP_FLAG_URGENT	The message has been flagged for urgent or special attention.
IMAP_FLAG_SEEN	The message has been read.

dwAccessMode

A value which specifies the access mode for the mailbox. It may be one of the following values:

Constant	Description
IMAP_ACCESS_READONLY	The mailbox has been selected in read-only mode. Messages may not be created in the mailbox, nor can message flags be modified.
IMAP_ACCESS_READWRITE	The mailbox has been selected in read-write mode. Messages may be modified by the client, and messages marked for deletion can be expunged.

dwReserved1

A reserved value that is undefined.

dwReserved2

A reserved value that is undefined.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

IMAPMAILBOXSTATUS Structure

This structure contains information about a mailbox.

```
typedef struct _IMAPMAILBOXSTATUS
{
    UINT    nMessages;
    UINT    nRecentMessages;
    UINT    nUnseenMessages;
    DWORD   dwMailboxUID;
    DWORD   dwNextMessageUID;
    DWORD   dwReserved;
} IMAPMAILBOXSTATUS, *LPIMAPMAILBOXSTATUS;
```

Members

nMessages

A value specifies the total number of messages in the mailbox.

nRecentMessages

A value which specifies the number of new messages that have recently arrived in the mailbox.

nUnseenMessages

A value which specifies the number of unread messages in the mailbox.

dwMailboxUID

A value which specifies a unique identifier for this mailbox which corresponds to the UIDVALIDITY value returned by the IMAP server. The actual value is determined by the server and should be considered opaque. The protocol specification requires that a mailbox's UID must not change unless the mailbox contents are modified or existing messages in the mailbox have been assigned new UIDs.

dwNextMessageUID

A value which specifies the predicted unique identifier that will be assigned to a new message in the mailbox. This corresponds to the UIDNEXT value returned by the IMAP server. The protocol specification requires that as long as the mailbox UID is unchanged, messages that are added to the mailbox will be assigned a UID greater than or equal to the next UID value.

dwReserved

A reserved value that is undefined.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

IMAPMESSAGE Structure

This structure contains information about a message.

```
typedef struct _IMAPMESSAGE
{
    UINT    nMessageId;
    DWORD   dwMessageUID;
    DWORD   dwSize;
    DWORD   dwFlags;
    DWORD   dwTimestamp;
    DWORD   dwReserved;
} IMAPMESSAGE, *LPIMAPMESSAGE;
```

Members

nMessageId

An integer value which identifies the message. The message identifier is only valid while the mailbox is selected and no messages marked for deletion have been expunged. To maintain a persistent identifier for the message, use a combination of the mailbox UID and message UID.

dwMessageUID

An integer value which specifies a unique identifier for this message. The actual value is determined by the server and should be considered opaque. If the client application stores the message UID on the local system, it should also store the UID for the mailbox that contains the message. If the mailbox UID changes, the message UID may no longer be valid.

dwSize

Specifies the size of the message in bytes.

dwFlags

A value which specifies one or more message flags. One or more of the following values may be specified:

Constant	Description
IMAP_FLAG_NONE	No value.
IMAP_FLAG_ANSWERED	The message has been answered.
IMAP_FLAG_DELETED	The message has been marked for deletion.
IMAP_FLAG_DRAFT	The message has not been completed and is marked as a draft copy.
IMAP_FLAG_URGENT	The message has been flagged for urgent or special attention.
IMAP_FLAG_RECENT	The message has been added to the mailbox recently.
IMAP_FLAG_SEEN	The message has been read.

dwTimestamp

An integer value which specifies the date and time that the message was created in the mailbox. The value is expressed as the number of seconds since midnight, 1 January 1970 and is the same value that is used for the standard C runtime library time functions. Note that the date and time used is the message's internal date from the mail server, not the value of the Date header field.

dwReserved

A reserved value that is undefined.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmapi10.lib

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

IMAPTRANSFERSTATUS Structure

This structure is used by the [ImapGetTransferStatus](#) function to return information about a file transfer in progress.

```
typedef struct _IMAPTRANSFERSTATUS
{
    UINT    nMessageId;
    DWORD   dwBytesTotal;
    DWORD   dwBytesCopied;
    DWORD   dwBytesPerSecond;
    DWORD   dwTimeElapsed;
    DWORD   dwTimeEstimated;
} IMAPTRANSFERSTATUS, *LPIMAPTRANSFERSTATUS;
```

Members

nMessageId

The message ID of the current message that is being transferred.

dwBytesTotal

The total number of bytes that will be transferred. If the file is being copied from the server to the local host, this is the size of the remote file. If the file is being copied from the local host to the server, it is the size of the local file. If the file size cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the file transfer is completed. This is based on the average number of bytes transferred per second.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

INITDATA Structure

This structure contains information about the SocketTools library when the initialization function returns.

```
typedef struct _INITDATA
{
    DWORD      dwSize;
    DWORD      dwVersionMajor;
    DWORD      dwVersionMinor;
    DWORD      dwVersionBuild;
    DWORD      dwOptions;
    DWORD_PTR  dwReserved1;
    DWORD_PTR  dwReserved2;
    TCHAR      szDescription[128];
} INITDATA, *LPINITDATA;
```

Members

dwSize

Size of this structure. This structure member must be set to the size of the structure prior to calling the initialization function. Failure to do so will cause the function to fail.

dwVersionMajor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the major version number for the library.

dwVersionMinor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the minor version number for the library.

dwVersionBuild

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the build number for the library.

dwOptions

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved1

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved2

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

szDescription

A null terminated string which describes the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

SECURITYCREDENTIALS Structure

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on

	what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that

will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A pointer to a null terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A pointer to a null terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A pointer to a null terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A pointer to a null terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
------------	-------------

CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
----	---

MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
----	--

ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.
------	--

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

A pointer to a null terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used for SSH connections

and should always be NULL when establishing a secure connection using SSL or TLS.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* member should be set to CREDENTIAL_STORE_FILENAME and the *lpzCertStore* member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS#12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the *lpzPassword* member or the library will be unable to access the certificate.

Note that the *lpzUserName* and *lpzPassword* members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

The TLS 1.1 and TLS 1.2 protocols are only supported on Windows 7, Windows Server 2008 R2 and later versions of the platform. If these options are specified and the application is running on Windows XP or Windows Vista, the protocol version will be downgraded to TLS 1.0 for backwards compatibility with those platforms.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

SECURITYINFO Structure

This structure contains information about a secure connection that has been established.

```
typedef struct _SECURITYINFO
{
    DWORD        dwSize;
    DWORD        dwProtocol;
    DWORD        dwCipher;
    DWORD        dwCipherStrength;
    DWORD        dwHash;
    DWORD        dwHashStrength;
    DWORD        dwKeyExchange;
    DWORD        dwCertStatus;
    SYSTEMTIME   stCertIssued;
    SYSTEMTIME   stCertExpires;
    LPCTSTR      lpszCertIssuer;
    LPCTSTR      lpszCertSubject;
    LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to **sizeof(SECURITYINFO)** prior to passing the address of this structure to the function. Note that if this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the protocol that was selected to establish the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The

	<p>correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.</p>
SECURITY_PROTOCOL_TLS10	<p>The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS11	<p>The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS12	<p>The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.</p>

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_CIPHER_RC2	The RC2 block cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
SECURITY_CIPHER_DES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher, using 56-bit

	keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher (also known as the Rijndael cipher) is a fixed block size cipher which use a key size of 128, 192 or 256 bits. This cipher is supported on Windows XP SP3 and later versions of the operating system.
SECURITY_CIPHER_SKIPJACK	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
SECURITY_CIPHER_BLOWFISH	The Blowfish block cipher was selected. This is a variable key length cipher up to 448 bits, using a 64-bit block size.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected. One of the following values will be returned:

Constant	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected. One of the following values will be returned:

--	--

Constant	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography. This key exchange algorithm is only supported on Windows XP SP3 and later versions of the operating system.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server. This member only has meaning for connections using the SSL or TLS protocols. One of the following values will be returned:

Constant	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertIssuer

A pointer to a string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertSubject

A pointer to a string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszFingerprint

A pointer to a string which contains a sequence of hexadecimal values that uniquely identify the server. This member is only used when a connection has been established using the Secure Shell (SSH) protocol.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Unicode: Implemented as Unicode and ANSI versions.

Mail Message Library

Compose and parse standard MIME formatted email messages.

Reference

- [Functions](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

File Name	CSMSGV10.DLL
Version	10.0.1468.2518
LibID	E6D79220-7873-4170-BF39-5D7A0049655C
Import Library	CSMSGV10.LIB
Dependencies	None
Standards	RFC 822, RFC 2045, RFC 2046, RFC 2047, RFC 2048

Overview

The Mail Message library provides an interface for composing and processing email messages and newsgroup articles which are structured according to the Multipurpose Internet Mail Extensions (MIME) standard. Using this library, an application can easily create complex messages which include multiple alternative content types, such as plain text and styled HTML text, file attachments and customized headers.

It is not required that the developer understand the complex MIME standard; a single function call can be used to create multipart message, complete with a styled HTML text body and support for international character sets. The Mail Message library can be easily integrated with the other mail related protocol libraries, making it extremely easy to create and process MIME formatted messages.

The library also includes an interface for managing a local message storage file that can be used to store and retrieve multiple messages. Functions are provided to open and create storage files, add, remove and extract messages from storage, and search the stored messages for specific header field values.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-

bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Mail Message Functions

Function	Description
MimeAppendMessageText	Append text to the body of the current message part
MimeAttachData	Attach the contents of a buffer to the specified message
MimeAttachFile	Attach a file to the specified message
MimeAttachImage	Attach an inline image to the specified message
MimeClearMessageText	Clear the body of the current message part
MimeCloseMessageStore	Close the specified message storage file
MimeCompareMessageText	Compare text in the body of the current message part
MimeComposeMessage	Compose a new message using the specified parameters
MimeCopyMessageStore	Duplicate the contents of the specified message store in a new file
MimeCreateMessage	Create a new empty message
MimeCreateMessageEx	Create a new message from the contents of a file or a message in memory
MimeCreateMessagePart	Create a new message part for the specified message
MimeCreateMessagePartEx	Create a new message part for the specified message
MimeDecodeText	Decode a base64 or quoted-printable encoded string
MimeDecodeTextEx	Decode a base64 or quoted-printable encoded string using a specified character set
MimeDeleteMessage	Delete the specified message
MimeDeleteMessageHeader	Delete the specified header field from the message
MimeDeleteMessagePart	Delete the specified message part
MimeDeleteStoredMessage	Remove the specified message from the message store
MimeEncodeText	Encode a string using base64 or quoted-printable encoding
MimeEncodeTextEx	Encode a string using base64 or quoted-printable encoding using a specified character set
MimeEnumAttachments	Enumerate all file attachments in the specified message
MimeEnumMessageHeaders	Enumerate all header fields in the current message part
MimeEnumMessageRecipients	Enumerate addresses of all message recipients
MimeExportMessage	Export the current message to a text file
MimeExportMessageEx	Export the current message to a file, clipboard or memory buffer
MimeExtractAllFiles	Extract all file attachments in the message and store them in the specified directory
MimeExtractFile	Extract the file attachment from the current message part
MimeExtractFileEx	Extract a file attachment from the message with additional options
MimeFindAttachment	Search for a file attachment in the specified message
MimeFindStoredMessage	Search for a message in the specified message store
MimeFormatDate	Return a standard RFC 822 formatted date string
MimeGetAllHeaders	Return the complete RFC 822 header values in a string buffer
MimeGetAllRecipients	Return a comma-separated list of recipient addresses in a string buffer
MimeGetAttachedFileName	Return the name of the file attachment for the current part
MimeGetContentDigest	Return encoded digest of message's content
MimeGetContentLength	Return the length of the current message part content

MimeGetErrorString	Return a description for the specified error code
MimeGetExportOptions	Return a bitmask that describes current message export options
MimeGetFileContentType	Return the content type for a specified file
MimeGetFirstMessageHeader	Return the first header field and value in the current message part
MimeGetLastError	Return the last error code
MimeGetMessageBoundary	Return the multipart message boundary string
MimeGetMessageDate	Return the date and time from the message header
MimeGetMessageHeader	Return the value of a specified header from the message
MimeGetMessageHeaderEx	Copy the value of specified header to a string buffer
MimeGetMessagePart	Return the current message part index
MimeGetMessagePartCount	Return the total number of message parts
MimeGetMessageSender	Return the email address of the message sender
MimeGetMessageSize	Return the size of the complete message in bytes
MimeGetMessageText	Return the text of the current message part
MimeGetMessageVersion	Return the MIME version from the message header
MimeGetNextMessageHeader	Return the next header field and value in the current message part
MimeGetStoredMessage	Retrieve a message from the specified message store
MimeGetStoredMessageCount	Return the number of messages in a message store
MimeImportMessage	Import a message from the specified text file
MimeImportMessageEx	Import a message from a file, clipboard or memory buffer
MimeInitialize	Initialize the library for use by the client
MimeLocalizeText	Localize Unicode text to ANSI using a specific character set
MimeOpenMessageStore	Open the specified message storage file
MimeParseAddress	Parse the specified email address
MimeParseBuffer	Parse the specified text and add to the current message
MimeParseDate	Parse the specified RFC 822 formatted date string
MimeParseHeader	Parse the specified text and add to message header
MimePurgeMessageStore	Purge all deleted messages from the specified message store
MimeReplaceStoredMessage	Replace the specified message in the message store
MimeResetMessage	Clear the specified message, deleting all message parts
MimeSetExportOptions	Specify a bitmask that describes current message export options
MimeSetFileContentType	Set the content type for a specific file name extension
MimeSetLastError	Set the last error code
MimeSetMessageDate	Set the current date in the header for the specified message
MimeSetMessageHeader	Create or update a header field in the specified message
MimeSetMessageHeaderEx	Create or update a header field, with additional options
MimeSetMessagePart	Set the current message part index for the specified message
MimeSetMessageText	Create or update the specified message body
MimeSetMessageVersion	Set the MIME version for the specified message
MimeStoreMessage	Store the specified message in a message store

MimeAppendMessageText Function

```
LONG WINAPI MimeAppendMessageText(  
    HMESSAGE hMessage,  
    LPCTSTR lpszText  
);
```

The **MimeAppendMessageText** function appends the specified text to the body of the current message part.

Parameters

hMessage

Handle to the message.

lpszText

A pointer to a string which specifies the text to be appended to the current message part.

Return Value

If the function succeeds, the return value is the number of bytes copied into the message. A return value of zero indicates that no text could be appended to the current message part. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeClearMessageText](#), [MimeCompareMessageText](#), [MimeGetMessageText](#),
[MimeSetMessageText](#)

MimeAttachData Function

```
BOOL WINAPI MimeAttachData(  
    HMESSAGE hMessage,  
    LPBYTE lpBuffer,  
    LONG cbBuffer,  
    LPCTSTR lpszContentName,  
    LPCTSTR lpszContentType,  
    DWORD dwOptions  
);
```

The **MimeAttachData** function attaches the contents of the buffer to the message.

Parameters

hMessage

Handle to the message.

lpBuffer

Pointer to a byte buffer which contains the data to be attached to the message. This parameter may be NULL, in which case no data is attached, but an additional empty message part will be created.

cbBuffer

An unsigned integer which specifies the number of bytes of data in the buffer pointed to by the *lpBuffer* parameter. If the *lpBuffer* parameter is NULL, this value must be zero.

lpszContentName

Pointer to a string which specifies a name for the data being attached to the message. This typically is used as a file name by the mail client to store the data in. If this parameter is NULL or an empty string then no name is defined and the data is attached as inline content. Note that if a file name is specified with a path, only the base name will be used.

lpszContentType

Pointer to a string which specifies the type of data being attached. The value must be a valid MIME content type. If this parameter is NULL or an empty string, then the buffer will be examined to determine what kind of data it contains. If there is only text characters, then the content type will be specified as "text/plain". If the buffer contains binary data, then the content type will be specified as "application/octet-stream", which is appropriate for any type of data.

dwOptions

A value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
MIME_ATTACH_DEFAULT	The data encoding is based on the content type. Text data is not encoded, and binary data is encoded using the standard base64 encoding algorithm.
MIME_ATTACH_BASE64	The data is always encoded using the standard base64 algorithm, even if the buffer only contains printable text characters.
MIME_ATTACH_UUCODE	The data is always encoded using the uuencode algorithm, even if the buffer only contains printable text characters.

MIME_ATTACH_QUOTED	The data is always encoded using the quoted-printable algorithm. This encoding should only be used if the data contains 8-bit text characters.
--------------------	--

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeAttachData** function allows an application to attach data to the message as either a file attachment or as inline content. The recipient of the message will see the attached data in the same way that they would see a file attached to the message using the **MimeAttachFile** function.

If the specified message is not a multipart message, it is marked as multipart and the attached file is appended to the message. If the message is already a multipart message, an additional part is created and the attachment is added to the message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeAttachFile](#), [MimeExportMessage](#), [MimeExtractFile](#), [MimeGetAttachedFileName](#), [MimeGetFileContentType](#), [MimeImportMessage](#), [MimeSetFileContentType](#)

MimeAttachFile Function

```
BOOL WINAPI MimeAttachFile(  
    HMESSAGE hMessage,  
    LPCTSTR lpszFileName,  
    DWORD dwOptions  
);
```

The **MimeAttachFile** function attaches the specified file to the message.

Parameters

hMessage

Handle to the message.

lpszFileName

Pointer to a string which specifies the name of the file to be attached to the message.

dwOptions

A value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
MIME_ATTACH_DEFAULT	The file attachment encoding is based on the file content type. Text files are not encoded, and binary files are encoded using the standard base64 encoding algorithm. This is the default option for file attachments.
MIME_ATTACH_BASE64	The file attachment is always encoded using the standard base64 algorithm, even if the attached file is a plain text file.
MIME_ATTACH_UUCODE	The file attachment is always encoded using the uuencode algorithm, even if the attached file is a plain text file.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

If the specified message is not a multipart message, it is marked as multipart and the attached file is appended to the message. If the message is already a multipart message, an additional part is created and the attachment is added to the message.

To attach data that is stored in a memory buffer rather than a file, use the **MimeAttachData** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

MimeAttachData, MimeExportMessage, MimeExtractFile, MimeGetAttachedFileName,
MimeGetFileContentType, MimeImportMessage, MimeSetFileContentType

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

MimeAttachImage Function

```
BOOL WINAPI MimeAttachImage(  
    HMESSAGE hMessage,  
    LPCTSTR lpszImageFile,  
    LPCTSTR lpszContentId  
);
```

The **MimeAttachImage** function attaches the specified file to the message as an inline image.

Parameters

hMessage

Handle to the message.

lpszFileName

Pointer to a string which specifies the name of the image file to be attached to the message. This parameter cannot be NULL and must specify the name of an existing file.

lpszContentId

Pointer to a string which specifies the content ID that is associated with the inline image. If this parameter is NULL, a random content ID string will be automatically generated.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeAttachImage** function enables an application to attach an inline image to the message. Unlike a normal file attachment, this function is designed to be used with HTML formatted email messages that display images attached to the message. The *lpszContentId* parameter specifies the content ID string that is used with the HTML image tag to reference that image.

If the specified message is not a multipart message, it is marked as multipart and the attached image file is appended to the message. If the message is already a multipart message, an additional part is created and the attachment is added to the message.

To attach regular files to the message, use the **MimeAttachFile** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeAttachData](#), [MimeAttachFile](#), [MimeComposeMessage](#)

MimeClearMessageText Function

```
BOOL WINAPI MimeClearMessageText(  
    HMESSAGE hMessage  
);
```

The **MimeClearMessageText** function deletes the text from the body of the current message part.

Parameters

hMessage

Handle to the message.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeAppendMessageText](#), [MimeCompareMessageText](#), [MimeGetMessageText](#), [MimeSetMessageText](#)

MimeCloseMessageStore Function

```
BOOL WINAPI MimeCloseMessageStore(  
    HMESSAGESTORE hStorage,  
    BOOL bPurgeMessages  
);
```

The **MimeCloseMessageStore** function closes the specified message store.

Parameters

hStorage

Handle to the message store.

bPurgeMessages

An integer value which specifies if deleted messages are purged from the message store. A non-zero value specifies that all messages marked for deletion will be removed from the message store. A value of zero specifies that deleted messages will not be removed from the store.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeCloseMessageStore** function closes the storage file that was previously opened, releasing all of the memory allocated for the message store and optionally purging all deleted messages. This function must be called when the application has finished accessing the messages in the message store. Failure to call this function will leave the storage file open and potentially locked by the process.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

See Also

[MimeCopyMessageStore](#), [MimeOpenMessageStore](#), [MimePurgeMessageStore](#)

MimeCompareMessageText Function

```
BOOL WINAPI MimeCompareMessageText(  
    HMESSAGE hMessage,  
    LONG* lpnOffset,  
    LPCTSTR lpszBuffer,  
    LONG cchBuffer,  
    BOOL bCaseSensitive  
);
```

The **MimeCompareMessageText** function compares a text string against the contents of the current message part.

Parameters

hMessage

Handle to the message.

lpnOffset

Pointer to a long integer which specifies the offset in the message at which to begin the comparison. This value will be updated when the function returns to indicate the offset position in the message where the comparison ended.

lpszBuffer

Pointer to a string buffer which contains the text that is to be compared against the body of the message.

cchBuffer

The number of characters in the buffer that should be compared against the body of the message.

bCaseSensitive

Boolean flag which specifies that the comparison should be case sensitive.

Return Value

If the text buffer matches the contents of the current message body, the function will return a non-zero value, and the *lpnOffset* argument will be set to position in the buffer where the match terminated. If the text buffer does not match, the function will return a value of zero, and the *lpnOffset* argument will be set to the position of the first non-matching character. The function will also return zero if one of the arguments is invalid. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeAppendMessageText](#), [MimeClearMessageText](#), [MimeGetMessageText](#), [MimeSetMessageText](#)

MimeComposeMessage Function

```
HMESSAGE WINAPI MimeComposeMessage(  
    LPCTSTR LpszFrom,  
    LPCTSTR LpszTo,  
    LPCTSTR LpszCc,  
    LPCTSTR LpszSubject,  
    LPCTSTR LpszMessageText,  
    LPCTSTR LpszMessageHTML,  
    UINT nCharacterSet,  
    UINT nEncodingType  
);
```

The **MimeComposeMessage** function creates a new message using the specified parameters.

Parameters

LpszFrom

A pointer to a string which specifies the sender's email address. This parameter may be NULL, in which case no sender address will be included in the message header.

LpszTo

A pointer to a string which specifies one or more recipient addresses. If multiple addresses are specified, each address must be separated by a comma. This parameter may be NULL, in which case no recipient addresses will be included in the message header.

LpszCc

A pointer to a string which specifies one or more addresses that will receive a copy of the message in addition to the listed recipients. If multiple addresses are specified, each address must be separated by a comma. This parameter may be NULL, in which case no carbon-copy addresses will be included in the message header.

LpszSubject

A pointer to a string which specifies the subject of the message. This parameter may be NULL, in which case no subject will be included in the message.

LpszMessageText

A pointer to a string which contains the body of the message as plain text. Each line of text contained in the string should be terminated with a carriage-return and linefeed (CRLF) pair, which is recognized as the end-of-line. If this parameter is NULL or points to an empty string, then the message will have an empty body unless the *LpszMessageHTML* parameter is not NULL.

LpszMessageHTML

A pointer to a string which contains the message using HTML formatting. If the *LpszMessageText* parameter is not NULL, then a multipart message will be created with both plain text and HTML text as the alternative. This allows mail clients to select which message body they wish to display. If the *LpszMessageText* argument is NULL or points to an empty string, then the message will only contain HTML. Although this is supported, it is not recommended because older mail clients may be unable to display the message correctly.

nCharacterSet

A numeric identifier which specifies the [character set](#) to use when composing the message. A value of zero specifies that the default UTF-8 character set should be used. It is recommended that you always use UTF-8 when composing a new message or creating a new message part.

nEncodingType

A numeric identifier which specifies the encoding type to use when composing the message. A value of zero specifies that default 7bit encoding should be used. The following values may also be used:

Constant	Description
MIME_ENCODING_7BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the most significant bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_ENCODING_8BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long and all bits are used. 8-bit encoding is typically used with multibyte character sets and is the default encoding used with Unicode text.
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.
MIME_ENCODING_BASE64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of data. This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text.

Return Value

If the function succeeds, the return value is a handle to the message. If the function fails, the return value is `INVALID_MESSAGE`. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeComposeMessage** function composes a new message and returns a handle which can be used to further modify or export the message. To create an empty message without any predefined header values, call the **MimeCreateMessage** function.

email addresses may be specified as simple addresses, or as commented addresses that include the sender's name or other information. For example, any one of these address formats are acceptable:

```
user@domain.tld
User Name <user@domain.tld>
user@domain.tld (User Name)
```

To specify multiple addresses, you should separate each address by a comma or semi-colon. Note that the *lpzFrom* parameter cannot specify multiple addresses, however it is permitted with the *lpzTo*, *lpzCc* and *lpzBcc* parameters.

To send a message that contains HTML, it is recommended that you provide both a plain text version of the message body and an HTML formatted version. While it is permitted to send a message that only contains HTML, some older mail clients may not be capable of displaying the message correctly. In some cases, anti-spam software will increase the spam score of messages that do not contain a plain text message body. This can result in your message being rejected or quarantined by the mail server.

Call the **MimeDeleteMessage** function to free the memory allocated for this message when it is no longer needed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeCreateMessage](#), [MimeCreateMessagePart](#), [MimeDeleteMessage](#), [MimeExportMessage](#), [MimeImportMessage](#), [MimeInitialize](#)

MimeCopyMessageStore Function

```
BOOL WINAPI MimeCopyMessageStore(  
    HMESSAGESTORE hStorage,  
    LPCTSTR lpszFileName  
);
```

The **MimeCopyMessageStore** function duplicates the contents of the specified message store in a new file.

Parameters

hStorage

Handle to the message store.

lpszFileName

A pointer to a string which specifies the name of the file that the messages will be copied to. This parameter cannot be NULL and must specify a valid file path and name.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeCopyMessageStore** function is used to create a copy of the specified message store in a new file. If the file does not exist, it will be created. If the file already exists, then the contents will be overwritten with the contents of the message store.

Messages that have been marked for deletion are not copied to the new message store file.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeCloseMessageStore](#), [MimeOpenMessageStore](#), [MimePurgeMessageStore](#)

MimeCreateMessage Function

```
HMESSAGE MimeCreateMessage();
```

The **MimeCreateMessage** function creates a new empty message.

Parameters

There are no parameters.

Return Value

If the function succeeds, the return value is a handle to the message. If the function fails, the return value is `INVALID_MESSAGE`. To get extended error information, call **MimeGetLastError**.

Remarks

To create a message with defined values for the sender, recipient, subject and body it is recommended that you use the **MimeComposeMessage** function.

Call the **MimeDeleteMessage** function to free the memory allocated for this message when it is no longer needed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

See Also

[MimeComposeMessage](#), [MimeCreateMessageEx](#), [MimeCreateMessagePart](#), [MimeDeleteMessage](#), [MimeExportMessage](#), [MimeImportMessage](#)

MimeCreateMessageEx Function

```
HMESSAGE WINAPI MimeCreateMessageEx(  
    DWORD dwImportMode,  
    DWORD dwImportOptions,  
    LPVOID lpvMessage,  
    DWORD dwMessageSize  
);
```

The **MimeCreateMessageEx** function creates a new message and imports the contents from a file or memory.

Parameters

dwImportMode

An unsigned integer which specifies how the message contents will be imported. It may be one of the following values:

Constant	Description
MIME_IMPORT_DEFAULT	The default import mode. If the <i>lpvMessage</i> parameter is NULL, the function will return a handle to an empty message. Otherwise, the <i>lpvMessage</i> parameter is a pointer to a string which specifies the name of a file that contains the message. The <i>dwMessageSize</i> parameter is ignored.
MIME_IMPORT_FILE	The <i>lpvMessage</i> parameter is a pointer to a string which specifies the name of a file that contains the message. If the file does not exist, or is not a regular text file, an error will occur. The <i>dwMessageSize</i> parameter is ignored.
MIME_IMPORT_CLIPBOARD	The contents of the message is imported from the system clipboard. The <i>lpvMessage</i> parameter is ignored. The <i>dwMessageSize</i> parameter is ignored.
MIME_IMPORT_MEMORY	The contents of the message is imported from a local buffer. The <i>lpvMessage</i> parameter must point to a byte array which contains the message to be imported. The <i>dwMessageSize</i> parameter specifies the number of bytes to copy from the buffer. If this value is zero, it is assumed that the end of the message data in the buffer is terminated with a null character and the length is calculated automatically.
MIME_IMPORT_HGLOBAL	The contents of the message is imported from a global memory buffer. The <i>lpvMessage</i> parameter must be a global memory handle which contains the message. The <i>dwMessageSize</i> parameter specifies the number of bytes to copy from the buffer. If this value is zero, it is assumed that the end of the message data in the buffer is terminated with a null character and the length is calculated automatically.

dwImportOptions

An unsigned integer which specifies how the message will be imported:

Constant	Description
MIME_OPTION_DEFAULT	The default import options. Currently this is the only valid value for this parameter and applications should always specify this constant.

lpvMessage

A pointer to a string, a byte buffer or a global memory handle. The *dwImportMode* parameter determines how this pointer is used by the function.

dwMessageSize

An unsigned integer value which specifies the size of the message to import. This parameter is only used when importing a message from a memory buffer. The message size is determined automatically when the message is imported from a file or the system clipboard.

Return Value

If the function succeeds, the return value is a handle to the message. If the function fails, the return value is `INVALID_MESSAGE`. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeCreateMessageEx** function creates a new message from the contents of a file or memory, and returns a handle to the parsed message. When the message is no longer needed by the application, it should call **MimeDeleteMessage** to free the memory allocated for the message and release the handle.

To create a new message with predefined header values and content, use the **MimeComposeMessage** function.

Example

The following example creates a message using an `HGLOBAL` handle that references a block of memory that contains an email message:

```
hMessage = MimeCreateMessageEx(MIME_IMPORT_HGLOBAL,  
                               MIME_OPTION_DEFAULT,  
                               (LPVOID)hgb1Message,  
                               0);  
  
if (hMessage != INVALID_MESSAGE)  
{  
    // The message has been successfully created  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeComposeMessage](#), [MimeCreateMessage](#), [MimeDeleteMessage](#), [MimeExportMessage](#), [MimeImportMessage](#)

MimeCreateMessagePart Function

```
INT WINAPI MimeCreateMessagePart(  
    HMESSAGE hMessage  
);
```

The **MimeCreateMessagePart** function creates a new part for the specified message. If this the first part created for a message that does not have the multipart content type specified, the message is marked as multipart and the header fields are updated.

Parameters

hMessage

Handle to the message.

Return Value

If the function succeeds, the return value is the new message part number. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

See Also

[MimeAttachFile](#), [MimeCreateMessage](#), [MimeDeleteMessagePart](#), [MimeGetMessagePart](#), [MimeGetMessagePartCount](#), [MimeSetMessagePart](#)

MimeCreateMessagePartEx Function

```
INT WINAPI MimeCreateMessagePartEx(  
    HMESSAGE hMessage,  
    UINT nCharacterSet,  
    UINT nEncodingType,  
    LPCTSTR LpszText  
);
```

The **MimeCreateMessagePartEx** function creates a new part for the specified message. If this the first part created for a message that does not have the multipart content type specified, the message is marked as multipart and the header fields are updated.

Parameters

hMessage

Handle to the message.

nCharacterSet

A numeric identifier which specifies the [character set](#) to use when composing the message. A value of zero specifies the character set should be the same character set used to initially compose the message. It is recommended that you always use UTF-8 when composing a new message or creating a new message part.

nEncodingType

A numeric identifier which specifies the encoding type to use when composing the message. A value of zero specifies that default 7bit encoding should be used. The following values may also be used:

Constant	Description
MIME_ENCODING_7BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the most significant bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_ENCODING_8BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long and all bits are used. 8-bit encoding is typically used with multibyte character sets and is the default encoding used with Unicode text.
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use Latin character sets such as ISO-8859-1, as well as those which use HTML.
MIME_ENCODING_BASE64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of

data. This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text.

lpzText

A pointer to a string which specifies the text to be included in the body of the new message part. If this parameter is NULL or points to an empty string, no text is added to the message part.

Return Value

If the function succeeds, the return value is the new message part number. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeAttachFile](#), [MimeComposeMessage](#), [MimeCreateMessage](#), [MimeCreateMessagePart](#), [MimeDeleteMessagePart](#), [MimeGetMessagePart](#), [MimeGetMessagePartCount](#), [MimeSetMessagePart](#)

MimeDecodeText Function

```
LONG WINAPI MimeDecodeText(  
    UINT nEncodingType,  
    LPCTSTR lpszInput,  
    LONG cchInput,  
    LPTSTR lpszOutput,  
    LONG cchOutput  
);
```

The **MimeDecodeText** function decodes a string which was previously encoded using base64 or quoted-printable encoding.

Parameters

nEncodingType

An integer value that specifies the encoding method used. It may be zero or one of the following values:

Constant	Description
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.
MIME_ENCODING_BASE64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of data. This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text. This is the default encoding type used by this function.

lpszInput

A pointer to a null terminated string which contains the encoded text. This parameter cannot be a NULL pointer.

cchInput

An integer value which specifies the number of characters of text in the input string which should be decoded. If this value is -1, the entire length of the string up to the terminating null will be decoded.

lpszOutput

A pointer to a string buffer that will contain the decoded text. This buffer must be large enough to store all of the characters in the decoded text, including the terminating null character. This parameter cannot be NULL.

cchOutput

An integer value which specifies the maximum number of characters which can be copied into the output string buffer. The buffer must be large enough to store all of the decoded text and terminating null character. This value must be greater than zero.

Return Value

If the input buffer can be successfully decoded, the return value is the length of the decoded output string. If the function returns zero, then no text was decoded and the output string buffer will be empty. If the function fails, the return value is `MIME_ERROR`. To get extended error information, call **MimeGetLastError**.

Remarks

This function provides a means to decode text that was previously encoded using either base64 or quoted-printable encoding. In most cases, it is not necessary to use this function because the message parser will automatically decode the message text if necessary.

This function and the **MimeEncodeText** function use the UTF-8 character set. If the Unicode version of this function is called, the output text will be decoded as UTF-8, then converted to UTF-16 and returned to the caller. If the ANSI version of this function is used, the decoded output will always be returned to the caller using the UTF-8 character set.

If an unsupported encoding type is specified, this function will return `MIME_ERROR` and the output text string will be empty. In most cases, it is preferable to use `MIME_ENCODING_BASE64` as the encoding method, with quoted-printable encoding only used for legacy support.

If the original text was encoded using a different character set, use the **MimeDecodeTextEx** function, which enables you to specify an alternate character set.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeDecodeTextEx](#), [MimeEncodeText](#), [MimeEncodeTextEx](#), [MimeGetMessageText](#), [MimeLocalizeText](#), [MimeSetMessageText](#)

MimeDecodeTextEx Function

```
LONG WINAPI MimeDecodeTextEx(  
    UINT nCharacterSet,  
    UINT nEncodingType,  
    DWORD dwReserved,  
    LPCTSTR lpszInput,  
    LONG cchInput,  
    LPTSTR lpszOutput,  
    LONG cchOutput  
);
```

The **MimeDecodeTextEx** function decodes a string which was previously encoded using base64 or quoted-printable encoding. This extended version of the function enables the caller to specify the character set for the original text.

Parameters

nCharacterSet

A numeric identifier which specifies the [character set](#) to use when decoding the input text. A value of zero specifies the character set is undefined and no Unicode text conversion is performed when the input string is decoded. If this value does not match the character set used when the text was originally encoded, the resulting output text may be invalid.

nEncodingType

An integer value that specifies the encoding method used. It may be zero or one of the following values:

Constant	Description
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.
MIME_ENCODING_BASE64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of data. This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text. This is the default encoding type used by this function.

dwReserved

An integer value reserved for internal use. This value must always be zero.

lpszInput

A pointer to a null terminated string which contains the encoded text. This parameter cannot be a NULL pointer.

cchInput

An integer value which specifies the number of characters of text in the input string which should be decoded. If this value is -1, the entire length of the string up to the terminating null will be decoded.

lpzOutput

A pointer to a string buffer that will contain the decoded text. This buffer must be large enough to store all of the characters in the decoded text, including the terminating null character. This parameter cannot be NULL.

cchOutput

An integer value which specifies the maximum number of characters which can be copied into the output string buffer. The buffer must be large enough to store all of the decoded text and terminating null character. This value must be greater than zero.

Return Value

If the input buffer can be successfully decoded, the return value is the length of the decoded output string. If the function returns zero, then no text was decoded and the output string buffer will be empty. If the function fails, the return value is `MIME_ERROR`. To get extended error information, call **MimeGetLastError**.

Remarks

This function provides a means to decode text that was previously encoded using either base64 or quoted-printable encoding. In most cases, it is not necessary to use this function because the message parser will detect which character set and encoding was used, then automatically decode the message text if necessary.

The value of the *nCharacterSet* parameter does not affect the resulting output text, it is only used when decoding the input text. If the Unicode version of this function is called, the output text will be converted to UTF-16 and returned to the caller. If the ANSI version of this function is used, the decoded output will always be returned to the caller using the UTF-8 character set.

If the *nCharacterSet* parameter is specified as `MIME_CHARSET_UTF16`, the encoding type must be `MIME_ENCODING_BASE64`. Other encoding methods are not supported for Unicode strings and will cause the function to fail. In most cases, it is preferable to use `MIME_ENCODING_BASE64` as the encoding method, with quoted-printable encoding only used for legacy support.

If an unsupported encoding type is specified, this function will return `MIME_ERROR` and the output text string will be empty. This function cannot be used to decode uuencoded text.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeDecodeText](#), [MimeEncodeText](#), [MimeEncodeTextEx](#), [MimeGetMessageText](#), [MimeLocalizeText](#), [MimeSetMessageText](#)

MimeDeleteMessage Function

```
BOOL WINAPI MimeDeleteMessage(  
    HMESSAGE hMessage  
);
```

The **MimeDeleteMessage** function deletes the specified message and releases the memory allocated for the header and body.

Parameters

hMessage

Handle to the message.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

Once the message has been deleted, the handle is invalid and should not be referenced.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

See Also

[MimeCreateMessage](#), [MimeDeleteMessagePart](#), [MimeResetMessage](#)

MimeDeleteMessageHeader Function

```
BOOL WINAPI MimeDeleteMessageHeader(  
    HMESSAGE hMessage,  
    LPCTSTR lpszHeader  
);
```

The **MimeDeleteMessageHeader** function deletes the specified header field from the message.

Parameters

hMessage

Handle to the message.

lpszHeader

Pointer to a string which specifies the header field that will be deleted.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeGetMessageHeader](#), [MimeSetMessageHeader](#)

MimeDeleteMessagePart Function

```
BOOL WINAPI MimeDeleteMessagePart(  
    HMESSAGE hMessage,  
    INT nMessagePart  
);
```

The **MimeDeleteMessagePart** function deletes the specified message part from the multipart message. The memory allocated for the message part is released.

Parameters

hMessage

Handle to the message.

nMessagePart

The message part index to delete.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

This function cannot be used to delete part zero, which is the main body of the message. Instead use the **MimeResetMessage** function to clear the entire message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeCreateMessagePart](#), [MimeCreateMessagePartEx](#), [MimeGetMessagePart](#),
[MimeGetMessagePartCount](#), [MimeResetMessage](#), [MimeSetMessagePart](#)

MimeDeleteStoredMessage Function

```
BOOL WINAPI MimeDeleteStoredMessage(  
    HMESSAGESTORE hStorage,  
    LONG nMessageId  
);
```

The **MimeDeleteStoredMessage** function removes the specified message from the message store.

Parameters

hStorage

Handle to the message store.

nMessageId

An integer value which identifies the message that is to be removed from the message store. Message numbers begin at one and increment for each message in the store.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeDeleteStoredMessage** function marks the specified message for deletion from the storage file. When the message store is closed or purged, the message is removed from the file. Once a message has been marked for deletion, it may no longer be referenced by the application. For example, you cannot access the contents of a message that has been deleted.

The message store must be opened with write access. This function will fail if you attempt to delete a message from a storage file that has been opened for read-only access. If the application needs to delete messages in the message store, it is recommended that the file be opened for exclusive access using the MIME_STORAGE_LOCK option.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

See Also

[MimeCloseMessageStore](#), [MimeFindStoredMessage](#), [MimeGetStoredMessage](#), [MimeGetStoredMessageCount](#), [MimePurgeMessageStore](#)

MimeEncodeText Function

```
LONG WINAPI MimeEncodeText(  
    UINT nEncodingType,  
    LPCTSTR lpszInput,  
    LONG cchInput,  
    LPTSTR lpszOutput,  
    LONG cchOutput  
);
```

The **MimeEncodeText** function encodes a string using base64 or quoted-printable encoding.

Parameters

nEncodingType

An integer value that specifies the encoding method used. It may be zero or one of the following values:

Constant	Description
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.
MIME_ENCODING_BASE64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of data. This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text. This is the default encoding type used by this function.

lpszInput

A pointer to a null terminated string which contains the encoded text. This parameter cannot be a NULL pointer.

cchInput

An integer value which specifies the number of characters of text in the input string which should be encoded. If this value is -1, the entire length of the string up to the terminating null will be encoded.

lpszOutput

A pointer to a string buffer that will contain the encoded text. This buffer must be large enough to store all of the characters in the encoded text, including the terminating null character. This parameter cannot be NULL.

cchOutput

An integer value which specifies the maximum number of characters which can be copied into the output string buffer. The buffer must be large enough to store all of the encoded text and terminating null character. This value must be greater than zero.

Return Value

If the input buffer can be successfully encoded, the return value is the length of the encoded output string. If the function returns zero, then no text was encoded and the output string buffer will be empty. If the function fails, the return value is `MIME_ERROR`. To get extended error information, call **MimeGetLastError**.

Remarks

This function provides a means to encode text using either base64 or quoted-printable encoding. It is not necessary to use this function to encode text when using the **MimeSetMessageText** function. The library will automatically encode message text which contains non-ASCII characters using the character set specified when the message is created.

This function and the **MimeDecodeText** function use the UTF-8 character set. If the Unicode version of this function is called, the input text will be converted to UTF-8 and then encoded. If the ANSI version of this function is used, the input text is expected to be UTF-8 or ASCII text. If you are encoding text which uses non-ASCII characters, it is recommended that you use the Unicode version of this function.

If an unsupported encoding type is specified, this function will return `MIME_ERROR` and the output text string will be empty. This function cannot be used to create uuencoded text. In most cases, it is preferable to use `MIME_ENCODING_BASE64` as the encoding method, with quoted-printable encoding only used for legacy support.

If you want the text to be encoded using a different character set, you can use the **MimeEncodeTextEx** function. In most cases, your application should use the default UTF-8 character set when encoding and decoding text which contains non-ASCII characters, rather than using specific character sets.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeDecodeText](#), [MimeDecodeTextEx](#), [MimeEncodeTextEx](#), [MimeGetMessageText](#), [MimeSetMessageText](#)

MimeEncodeTextEx Function

```
LONG WINAPI MimeEncodeTextEx(  
    UINT nCharacterSet,  
    UINT nEncodingType,  
    DWORD dwReserved,  
    LPCTSTR lpszInput,  
    LONG cchInput,  
    LPTSTR lpszOutput,  
    LONG cchOutput  
);
```

The **MimeEncodeTextEx** function encodes a string using base64 or quoted-printable encoding. This extended version of the function enables the caller to specify the character set for the input text.

Parameters

nCharacterSet

A numeric identifier which specifies the [character set](#) to use when encoding the input text. A value of zero specifies the character set is undefined and no Unicode text conversion is performed when the input string is encoded.

nEncodingType

An integer value that specifies the encoding method used. It may be one of the following values:

Constant	Description
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.
MIME_ENCODING_BASE64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of data. This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text.

dwReserved

An integer value reserved for internal use. This value must always be zero.

lpszInput

A pointer to a null terminated string which contains the encoded text. This parameter cannot be a NULL pointer.

cchInput

An integer value which specifies the number of characters of text in the input string which should be encoded. If this value is -1, the entire length of the string up to the terminating null will be encoded.

lpzOutput

A pointer to a string buffer that will contain the encoded text. This buffer must be large enough to store all of the characters in the encoded text, including the terminating null character. This parameter cannot be NULL.

cchOutput

An integer value which specifies the maximum number of characters which can be copied into the output string buffer. The buffer must be large enough to store all of the encoded text and terminating null character. This value must be greater than zero.

Return Value

If the input buffer can be successfully encoded, the return value is the length of the encoded output string. If the function returns zero, then no text was encoded and the output string buffer will be empty. If the function fails, the return value is `MIME_ERROR`. To get extended error information, call **MimeGetLastError**.

Remarks

This function provides a means to encode text using either base64 or quoted-printable encoding. It is not necessary to use this function to encode text when using the **MimeSetMessageText** function. The library will automatically encode message text which contains non-ASCII characters using the character set specified when the message is created.

If the *nCharacterSet* parameter is non-zero, the function will encode the text using the specified character set. If the Unicode version of this function is called, the input text is converted to ANSI using the code page associated with the character set. If the ANSI version of this function is called, the input text is converted to Unicode using the system default code page, and then back to ANSI using the specified character set.

If the *nCharacterSet* parameter specifies the `MIME_CHARSET_UTF16` character set, you must specify `MIME_ENCODING_BASE64` as the encoding method. Other encoding methods are not supported for Unicode strings and will cause the function to fail. It is not recommended you encode text as UTF-16 unless there is a specific requirement to use that character set.

It is recommended that you use the `MIME_CHARSET_UTF8` character set whenever possible. It is capable of encoding all Unicode code points, and is a standard for virtually all modern Internet applications. In most cases, it is preferable to use `MIME_ENCODING_BASE64` as the encoding method, with quoted-printable encoding only used for legacy support.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeDecodeText](#), [MimeEncodeText](#), [MimeEncodeTextEx](#), [MimeGetMessageText](#), [MimeSetMessageText](#)

MimeEnumAttachments Function

```
INT WINAPI MimeEnumAttachments(  
    HMESSAGE hMessage,  
    LPCTSTR* lpFileNames,  
    INT nMaxFiles  
);
```

The **MimeEnumAttachments** function enumerates all of the file attachments in the specified message.

Parameters

hMessage

Handle to the message.

lpFileNames

A pointer to an array of null-terminated strings that contain the names of the files attached to the message. If this parameter is NULL, the function will only return the number of files attached to the message.

nMaxFiles

An integer value that specifies the maximum size of the array of string pointers specified by the *lpFileNames* parameter. If this value is zero, the *lpFileNames* parameter is ignored and the function will only return the number of files attached to the message.

Return Value

If the function succeeds, the return value is the number of files attached to the message. If the message does not contain any file attachments, this function will return a value of zero. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

This function can be used to obtain the names of the files attached to the message. If this function is used with languages other than C/C++, it is important to note that the *lpFileNames* parameter is not a pointer to a string, but rather a pointer to an array of pointers to null-terminated strings. It is not recommended that you use this function with languages that do not support C-style arrays. For example, Visual Basic represents arrays as SAFEARRAY structures and is not compatible with this function.

Example

```
LPCTSTR lpszFiles[MAXFILES];  
  
INT nFiles = MimeEnumAttachments(hMessage, lpszFiles, MAXFILES);  
if (nFiles == MIME_ERROR)  
{  
    DWORD dwError = MimeGetLastError();  
    _tprintf(_T("Unable to enumerate attachments, error 0x%08lx\n"), dwError);  
    return;  
}  
  
_tprintf(_T("There are %d files attached to the message\n"), nFiles);  
for (INT nIndex = 0; nIndex < nFiles; nIndex++)  
{  
    BOOL bResult = MimeExtractFileEx(hMessage, -1, lpszFiles[nIndex], NULL, 0);
```

```
    if (bResult)
        _tprintf(_T("%d: extracted attachment \"%s\"\n"), nIndex,
lpszFiles[nIndex]);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeAttachFile](#), [MimeExportMessage](#), [MimeExtractAllFiles](#), [MimeExtractFileEx](#),
[MimeGetAttachedFileName](#), [MimeImportMessage](#)

MimeEnumMessageHeaders Function

```
INT WINAPI MimeEnumMessageHeaders(  
    HMESSAGE hMessage,  
    LPCTSTR* lpHeaderList,  
    INT nMaxHeaders  
);
```

The **MimeEnumMessageHeaders** function returns a list of pointers to all header field names in the current message part. This can be used in conjunction with the **MimeGetMessageHeader** function to retrieve the values for every header in the message.

Parameters

hMessage

Handle to the message.

lpHeaderList

Pointer to an array of pointers to null terminated header field names. If this parameter is NULL, the function only returns the number of headers in the current message part.

nMaxHeaders

The maximum number of header fields which may be returned in the *lpHeaderList* parameter.

Return Value

If the function succeeds, the return value is the total number of headers that are defined in the current message part. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

The values returned in the header list array must not be directly modified by the application. There is no specific order in which the header fields are enumerated by this function. The header fields from an imported message may not be returned in the same order as which they appear in the message. An application should never make an assumption about the order in which one or more header fields are defined.

Example

```
// Determine the total number of headers in the current  
// message part  
  
nHeaders = MimeEnumMessageHeaders(hMessage, NULL, 0);  
if (nHeaders > 0)  
{  
  
    // Allocate memory for the list of headers  
  
    lpHeaderList = (LPCTSTR *)malloc(nHeaders * sizeof(LPCTSTR));  
    assert(lpHeaderList != NULL);  
  
    // Retrieve the list of headers in the current  
    // message part, and get their values  
  
    MimeEnumMessageHeaders(hMessage, lpHeaderList, nHeaders);  
    for (nIndex = 0; nIndex < nHeaders; nIndex++)  
    {
```

```
LPCTSTR lpszValue;  
lpszValue = MimeGetMessageHeader(hMessage, lpHeaderList[nIndex]);  
assert(lpszValue != NULL);  
  
printf("%s: %s\n", lpHeaderList[nIndex], lpszValue);  
  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeGetFirstMessageHeader](#), [MimeGetMessageHeader](#), [MimeGetNextMessageHeader](#),
[MimeSetMessageHeader](#)

MimeEnumMessageRecipients Function

```
INT WINAPI MimeEnumMessageRecipients(  
    HMESSAGE hMessage,  
    LPCTSTR LpszExtraAddress,  
    LPTSTR lpBuffer,  
    LPDWORD LpcchBuffer  
);
```

The **MimeEnumMessageRecipients** function returns a null-terminated list of strings which contain the email address of each recipient for the specified message.

Parameters

hMessage

Handle to the message.

lpszExtraAddress

A pointer to a string which contains one or more additional email addresses that should be included in the list, in addition to those found in the message. If more than one address is specified, each address should be separated by a comma. This parameter may be NULL if there are no extra addresses to include in the recipient list.

lpBuffer

Pointer to buffer which will contain zero or more null-terminated strings. The end of the string list is indicated by an additional terminating null. If this parameter is NULL, the function will calculate the minimum number of bytes required to store the addresses and return the value in the *lpcbBuffer* parameter.

lpcbBuffer

A pointer to an unsigned integer which should be initialized to the maximum number of characters that can be copied into the buffer specified by the *lpBuffer* parameter. When the function returns, it will be updated to contain the actual number of characters copied into the buffer. If the *lpBuffer* parameter is NULL, then this value will contain the minimum number of characters required to store all of the recipient addresses in the current message.

Return Value

If the function succeeds, the return value is the total number of recipients for the current message. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeEnumMessageRecipients** function returns a list of recipient email addresses for the specified message, with each address being terminated by a null character. The end of the list is indicated by an additional null character. To determine the size of the buffer you should pass to this function, you can specify the *lpBuffer* parameter as NULL and initialize the value of the *lpcbBuffer* parameter to zero.

This function is primarily designed for use with C/C++ and languages that can easily use C-style strings and pointers to string values. In many cases, it may be preferable to use the **MimeGetAllRecipients** function which returns a comma-separated list of recipient addresses in a string buffer.

Example

```

LPTSTR lpRecipients = NULL;
DWORD cchRecipients = 0;
INT nRecipients = 0;

// Determine the number of characters that should be allocated to store
// all of the recipient addresses in the current message

nRecipients = MimeEnumMessageRecipients(hMessage,
                                         NULL,
                                         NULL,
                                         &cchRecipients);

// Allocate the memory for the string buffer that will contain all
// of the recipient addresses and call MimeEnumMessageRecipients
// again to store those addresses in the buffer

if (nRecipients > 0 && cchRecipients > 0)
{
    lpRecipients = (LPTSTR)LocalAlloc(LPTR, cchRecipients * sizeof(TCHAR));
    if (lpRecipients == NULL)
        return; // Virtual memory exhausted

    nRecipients = MimeEnumMessageRecipients(hMessage,
                                             NULL,
                                             lpRecipients,
                                             &cchRecipients);
}

// Move through the buffer, processing each recipient address
// that was returned

if (nRecipients > 0)
{
    LPTSTR lpszAddress = lpRecipients;
    INT cchAddress;

    while (lpszAddress != NULL)
    {
        if ((cchAddress = lstrlen(lpszAddress)) == 0)
            break;

        // lpszAddress specifies a recipient address
        // Advance to the next address string in the buffer
        lpszAddress += cchAddress + 1;
    }
}

if (lpRecipients)
{
    LocalFree((HLOCAL)lpRecipients);
    lpRecipients = NULL;
}

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeGetAllRecipients](#), [MimeGetFirstMessageHeader](#), [MimeGetMessageHeader](#),
[MimeGetNextMessageHeader](#), [MimeSetMessageHeader](#)

MimeExportMessage Function

```
BOOL WINAPI MimeExportMessage(  
    HMESSAGE hMessage,  
    LPCTSTR LpszFileName  
);
```

The **MimeExportMessage** function writes the specified message to a file. If the file does not exist, it will be created. If it does exist, it will be overwritten with the contents of the message.

Parameters

hMessage

Handle to the message.

lpszFileName

Pointer to a string which specifies the name of the file to be created from the message. If this parameter is NULL, the contents of the message will be copied to the system clipboard.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

This function is provided for backwards compatibility with previous versions of the API. To export the contents of the message to a buffer in memory, use the **MimeExportMessageEx** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeCreateMessage](#), [MimeExportMessageEx](#), [MimeImportMessage](#), [MimeImportMessageEx](#), [MimeSetExportOptions](#)

MimeExportMessageEx Function

```
BOOL WINAPI MimeExportMessageEx(  
    HMESSAGE hMessage,  
    DWORD dwExportMode,  
    DWORD dwExportOptions,  
    LPVOID lpvMessage,  
    LPDWORD lpdwMessageSize  
);
```

The **MimeExportMessageEx** function exports the message to a file, the system clipboard or global memory buffer.

Parameters

hMessage

Handle to the message.

dwExportMode

An unsigned integer which specifies how the message contents will be exported. It may be one of the following values:

Constant	Description
MIME_EXPORT_DEFAULT	The default export mode. If the <i>lpvMessage</i> parameter is NULL, then the contents of the message will be copied to the system clipboard. Otherwise, the <i>lpvMessage</i> parameter is a pointer to a string which specifies the name of a file to store the message in.
MIME_EXPORT_FILE	The <i>lpvMessage</i> parameter is a pointer to a null-terminated string which specifies the name of a file to store the message in. If the file does not exist, it will be created. Otherwise the file will be overwritten with the contents of the message.
MIME_EXPORT_CLIPBOARD	The contents of the message is copied to the system clipboard. The <i>lpvMessage</i> parameter is ignored.
MIME_EXPORT_MEMORY	The contents of the message is copied to a local buffer. The <i>lpvMessage</i> parameter must point to a byte array which will contain the message contents when the function returns. The <i>lpdwMessageSize</i> parameter must be initialized to the maximum size of the buffer and will contain the number of bytes copied to the buffer when the function returns.
MIME_EXPORT_HGLOBAL	The contents of the message is stored in a global memory buffer. The <i>lpvMessage</i> parameter must point to a global memory handle which will reference the message buffer when the function returns. The <i>lpdwMessageSize</i> parameter will contain the number of bytes allocated for the message. The client application is responsible for releasing the memory handle when the message contents are no longer needed.

dwExportOptions

An unsigned integer which specifies how the message will be exported. The following values may be combined using a bitwise Or operator:

Constant	Description
MIME_OPTION_DEFAULT	The default export options. The headers for the message are written out in a specific consistent order, with custom headers written to the end of the header block regardless of the order in which they were set or imported from another message. If the message contains Bcc, Received, Return-Path, Status or X400-Received header fields, they will not be exported.
MIME_OPTION_KEEORDER	The original order in which the message header fields were set or imported are preserved when the message is exported.
MIME_OPTION_ALLHEADERS	All headers, including the Received, Return-Path, Status and X400-Received header fields will be exported. Normally these headers are not exported because they are only used by the mail transport system. This option can be useful when exporting a message to be stored on the local system, but should not be used when exporting a message to be delivered to another user.
MIME_OPTION_NOHEADERS	When exporting a message, the main header block will not be included. This can be useful when creating a multipart message for services which expect MIME formatted data, such as HTTP POST requests. This option should never be used for email messages being submitted using SMTP.

lpvMessage

A pointer to a string, a byte buffer or a global memory handle. The *dwExportMode* parameter determines how this pointer is used by the function.

lpdwMessageSize

A pointer to an unsigned integer value which will contain the size of the message when the function exits. This parameter may be NULL if you do not require this information, except if the *dwExportMode* parameter is MIME_EXPORT_MEMORY. In this case, the parameter must point to a value initialized with the maximum size of the byte buffer that has been passed to the function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

If the MIME_EXPORT_MEMORY mode is used to export the message to a pre-allocated buffer, the *lpdwMessageSize* parameter must be initialized to the size of the buffer before being passed to the function by reference. If this value is not properly initialized by the application, it can result in an error or a general protection exception. If the function succeeds, the message will be copied to

the buffer. If the function fails, the previous contents of the buffer will not be preserved. It is not guaranteed that the buffer will be terminated with a null byte. If the buffer is not large enough to store the entire message, the function will fail. The **MimeGetMessageSize** function can be used to determine the minimum size of the buffer required to store the message.

If the MIME_EXPORT_HGLOBAL mode is used, a global memory handle will be returned to the caller which contains the message contents. This handle must be dereferenced using the **GlobalLock** function. No changes should be made to this copy of the message. If you wish to modify the contents of the message, allocate a local buffer and copy the message contents, or use the MIME_EXPORT_MEMORY option instead. Your application is responsible for calling **GlobalUnlock** and **GlobalFree** to unlock and free the handle when it is no longer needed.

Example

The following example exports the contents of a message to a global memory buffer:

```
HGLOBAL hgb1Message = NULL;
DWORD dwMessageSize = 0;

bResult = MimeExportMessageEx(hMessage,
                              MIME_EXPORT_HGLOBAL,
                              MIME_OPTION_DEFAULT,
                              &hgb1Message,
                              &dwMessageSize);

if (bResult)
{
    LPBYTE lpMessage = (LPBYTE)GlobalLock(hgb1Message);

    if (lpMessage)
    {
        // Process the contents of the message
    }

    GlobalUnlock(hgb1Message);
    GlobalFree(hgb1Message);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeCreateMessage](#), [MimeExportMessage](#), [MimeGetMessageSize](#), [MimeImportMessage](#), [MimeImportMessageEx](#), [MimeSetExportOptions](#)

MimeExtractAllFiles Function

```
INT WINAPI MimeExtractAllFiles(  
    HMESSAGE hMessage,  
    LPCTSTR lpszDirectory,  
    DWORD dwReserved  
);
```

The **MimeExtractAllFiles** function extracts all of the file attachments in a message and stores them in the specified directory.

Parameters

hMessage

Handle to the message.

lpszDirectory

A pointer to a string which specifies the name of the directory where the file attachments should be stored. If this parameter is NULL or points to an empty string, the attached files will be stored in the current working directory on the local system.

dwReserved

An unsigned integer value that is reserved for future use. This parameter must always have a value of zero.

Return Value

If the function succeeds, the return value is the number of file attachments which were extracted from the message. If the message does not contain any file attachments, this function will return a value of zero. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

This function will extract all of the files that are attached to the message and store them in the specified directory. The directory must exist and the current user must have the appropriate permissions to create files there. If a file with the same name as the attachment already exists, it will be overwritten with the contents of the attachment. If the file attachment was encoded using base64 or uuencode, this function will automatically decode the contents of the attachment.

To determine the file names for each of the attachments in a message, use the **MimeEnumAttachments** function. To store a file attachment on the local system using a name that is different than the file name of the attachment, use the **MimeExtractFile** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeAttachFile](#), [MimeEnumAttachments](#), [MimeExportMessage](#), [MimeExtractFileEx](#), [MimeGetAttachedFileName](#), [MimeImportMessage](#)

MimeExtractFile Function

```
BOOL WINAPI MimeExtractFile(  
    HMESSAGE hMessage,  
    LPCTSTR lpszFileName  
);
```

The **MimeExtractFile** function extracts a file attachment from the current message part and stores it on the local system.

Parameters

hMessage

Handle to the message.

lpszFileName

A pointer to a string which specifies the name of a file on the local system.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

This function will store the contents of a file attachment in the current message part to the specified file on the local system. If a path is specified as part of the file name, it must exist and the current user must have the appropriate permissions to create the file. If a file with the same name already exists, it will be overwritten with the contents of the attachment. If the file attachment was encoded using base64 or uuencode, this function will automatically decode the contents of the attachment.

To determine if the current message part contains a file attachment, use the **MimeGetAttachedFileName** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeAttachFile](#), [MimeExportMessage](#), [MimeExtractFileEx](#), [MimeGetAttachedFileName](#), [MimeImportMessage](#)

MimeExtractFileEx Function

```
BOOL WINAPI MimeExtractFileEx(  
    HMESSAGE hMessage,  
    INT nMessagePart,  
    LPCTSTR lpszAttachment,  
    LPCTSTR lpszFileName,  
    DWORD dwReserved  
);
```

The **MimeExtractFileEx** function extracts a file attachment from the message and stores it on the local system.

Parameters

hMessage

Handle to the message.

nMessagePart

An integer value that specifies the message part that contains the file attachment. This value may be -1, in which case the current message part will be used, or the function will search for an attachment with a file name that matches the value of the *lpszAttachment* parameter.

lpszAttachment

A pointer to a string that specifies the file name for the attachment in the message. If the file name of the attachment is not known, this parameter can be NULL or point to an empty string. This parameter is ignored if the *nMessagePart* parameter has a value greater than -1.

lpszFileName

A pointer to a string that specifies the name of a file on the local system. If this parameter is NULL or points to an empty string, the value of the *lpszAttachment* parameter will specify the name of the file in the current working directory. If both the *lpszAttachment* and *lpszFileName* parameters are NULL, the function will fail.

dwReserved

An unsigned integer value that is reserved for future use. This parameter must always have a value of zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

This function will store the contents of a file attachment in the current message part to the specified file on the local system. If a path is specified as part of the file name, it must exist and the current user must have the appropriate permissions to create the file. If a file with the same name already exists, it will be overwritten with the contents of the attachment. If the file attachment was encoded using base64 or uuencode, this function will automatically decode the contents of the attachment.

If the *nMessagePart* parameter has a value of -1 and the *lpszAttachment* parameter specifies a file name, this function will search the entire message for an attachment with the same file name. The search is not case-sensitive, however it must match the attachment file name completely. This function will not match partial file names or names that include wildcard characters. If a match is

found, the contents of that attachment will be stored in the file specified by the *lpzFileName* parameter.

If the *nMessagePart* parameter has a value of -1 and the *lpzAttachment* parameter is NULL or points to an empty string, then the attachment in the current message part will be stored in the specified file. If the current message part does not contain a file attachment, this function will fail. Calling this function in this manner is effectively the same as calling the **MimeExtractFile** function.

To extract all of the files attached to a message in a single function call, use the **MimeExtractAllFiles** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeAttachFile](#), [MimeEnumAttachments](#), [MimeExportMessage](#), [MimeExtractAllFiles](#), [MimeExtractFile](#), [MimeFindAttachment](#), [MimeImportMessage](#)

MimeFindAttachment Function

```
INT WINAPI MimeFindAttachment(  
    HMESSAGE hMessage,  
    LPCTSTR lpszFileName  
);
```

The **MimeFindAttachment** function searches the message for an attachment with the specified file name.

Parameters

hMessage

Handle to the message.

lpszFileName

Pointer to a string that specifies the name of the file attachment to search for. This parameter should only specify a base file name; it should not include a file path and cannot be NULL.

Return Value

If the function succeeds, the return value is the message part number that contains an attachment that matches the specified file name. If the message does not contain an attachment with the specified file name, the function will return MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

This function will search the message for a attachment that matches the specified file name. The search is not case-sensitive, however it must match the attachment file name completely. This function will not match partial file names or names that include wildcard characters. To obtain a list of all of the files attached to a message, use the **MimeEnumAttachments** function.

Example

```
// The name of the file attachment to search for  
LPCTSTR lpszFileName = _T("MyProject.docx");  
  
// Search for the attached file and store it on the local system  
INT nMessagePart = MimeFindAttachment(hMessage, lpszFileName);  
if (nMessagePart != MIME_ERROR)  
{  
    MimeSetMessagePart(hMessage, nMessagePart);  
  
    if (MimeExtractFile(hMessage, lpszFileName) != MIME_ERROR)  
        _tprintf(_T("Saved file attachment %s\n"), lpszFileName);  
    else  
    {  
        _tprintf(_T("Unable to save file attachment %s\n"), lpszFileName);  
        return;  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeAttachFile](#), [MimeExtractAllFiles](#), [MimeExtractFile](#), [MimeExtractFileEx](#),
[MimeGetAttachedFileName](#)

MimeFindStoredMessage Function

```
LONG WINAPI MimeFindStoredMessage(  
    HMESSAGESTORE hStorage,  
    LONG nMessageId,  
    LPCTSTR lpszHeaderName,  
    LPCTSTR lpszHeaderValue,  
    DWORD dwOptions  
);
```

The **MimeFindStoredMessage** function searches for a message in the specified message store.

Parameters

hStorage

Handle to the message store.

nMessageId

An integer value which specifies the message number that should be used when starting the search. The first message in the message store has a value of one.

lpszHeaderField

A pointer to the string which specifies the name of the header field that should be searched. The header field name is not case sensitive. This parameter cannot be NULL.

lpszHeaderValue

A pointer to the string which specifies the header value that should be searched for. The search options can be used to specify if the search is case-sensitive, and whether the search should return partial matches to the string. This parameter cannot be NULL.

dwOptions

A value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
MIME_SEARCH_DEFAULT	Perform a complete match against the specified header value. The comparison is not case-sensitive.
MIME_SEARCH_CASE_SENSITIVE	The header field value comparison will be case-sensitive. Note that this does not affect header field names. Matches for header names are always case-insensitive.
MIME_SEARCH_PARTIAL_MATCH	Perform a partial match against the specified header value. It recommended that this option be used when searching for matches to email addresses.
MIME_SEARCH_DECODE_HEADERS	Decode any encoded message headers before comparing them to the specified value. This option can increase the amount of time required to search the message store and should only be used when necessary.

Return Value

If the function succeeds, the return value is the number for the message which matches the search criteria. If the function fails, the return value is `MIME_ERROR`. To get extended error information, call `MimeGetLastError`.

Remarks

The `MimeFindStoredMessage` function is used to search the message store for a message which matches a specific header field value. For example, it can be used to find every message which is addressed to a specific recipient or has a subject which matches a particular string value.

Example

```
HMESSAGE hMessage = INVALID_MESSAGE;
LPCTSTR lpszHeader = _T("From");
LPCTSTR lpszAddress = _T("jsmith@example.com");
LONG nMessageId = 1;

// Begin searching for messages from the specified sender
while (nMessageId != MIME_ERROR)
{
    nMessageId = MimeFindStoredMessage(hStorage,
                                       nMessageId,
                                       lpszHeader,
                                       lpszAddress,
                                       MIME_SEARCH_PARTIAL_MATCH);

    if (nMessageId != MIME_ERROR)
    {
        // Get a handle to the message that was found
        hMessage = MimeGetStoredMessage(hStorage, nMessageId, 0);

        if (hMessage != INVALID_MESSAGE)
        {
            // Store the message in a file
            TCHAR szFileName[MAX_PATH];
            BOOL bExported;

            // Create a filename based on the message number
            wsprintf(szFileName, _T("msg%05ld.tmp"), nMessageId);

            // Export the message to a file
            bExported = MimeExportMessage(hMessage, szFileName);
        }

        // Increase the message ID to resume the search at the next message
        nMessageId++;
    }
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

MimeFormatDate Function

```
LPTSTR WINAPI MimeFormatDate(  
    LONG nSeconds,  
    LONG nTimezone,  
    LPTSTR lpszDate,  
    INT cchMaxDate  
);
```

The **MimeFormatDate** converts the specified date, expressed as the number of seconds since 1 January 1970, into a string compatible with the RFC 822 standard format for email messages.

Parameters

nSeconds

A long integer which specifies the number of seconds since 1 January 1970 00:00:00 UTC. This date is commonly called the epoch, and is the base date used by the standard C time functions. If the value of this parameter is zero, the current date and time is used.

nTimezone

A pointer to a long integer which is set to the difference in seconds between the specified date's timezone and Coordinated Universal Time. A value of zero specifies Coordinated Universal Time (UTC), while a positive value specifies a timezone west of UTC and a negative value specifies a timezone east of UTC. For example, Eastern Standard Time would be specified as 18000 and Pacific Standard Time would be 28800.

lpszDate

A pointer to a string buffer which will contain the formatted date. This parameter cannot be a NULL pointer.

cchMaxDate

The maximum number of characters, including the terminating null character, which may be copied into the date string buffer.

Return Value

If the function succeeds, a pointer to the date string buffer is returned. If the function fails, a NULL pointer is returned. To get extended error information, call **MimeGetLastError**.

Remarks

The date string is returned in a standard format as outlined in RFC 822, the document which describes the basic structure of Internet email messages. This format is as follows:

www, dd mmm yyyy hh:mm:ss [-]zzzz

Each part of the date string is defined as follows:

Format	Description
www	Weekday
dd	Day
mmm	Month
yyyy	Year
hh	Hour (24-hour clock)

mm	Minutes
ss	Seconds
zzzz	Timezone

The weekday and month are displayed using standard three-character English abbreviations. The timezone is displayed as the difference (in hours and minutes) between the specified timezone and Coordinated Universal Time. For example, if the timezone is eight hours west of Coordinated Universal Time, the *nTimezone* value would be 28800. This would be displayed as -0800 in the formatted date string.

Note that the format of the date string is defined by the RFC 822 standard, and is not affected by localization settings on the host system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeParseDate](#), [MimeGetMessageDate](#), [MimeSetMessageDate](#)

MimeGetAllHeaders Function

```
INT WINAPI MimeGetAllHeaders(  
    HMESSAGE hMessage,  
    LPTSTR LpszHeaders,  
    INT nMaxLength  
);
```

The **MimeGetAllHeaders** function returns the complete RFC 822 header values in a string buffer.

Parameters

hMessage

Handle to the message.

lpszHeaders

Pointer to string buffer which will contain the header values for the specified message. This parameter may be NULL, in which case the function will calculate the number of characters needed to store the complete header block.

nMaxLength

An integer value which specifies the maximum number of characters that can be stored in the *lpszHeaders* string. If the *lpszHeaders* parameter is NULL, this value must be zero. If the *lpszHeaders* parameter is not NULL, this value must be large enough to store the entire list of addresses.

Return Value

If the function succeeds and the *lpszHeaders* parameter is NULL, the return value is the minimum number of characters that should be allocated to store all of the header values, including the terminating null character. If the *lpszHeaders* parameter is not NULL, then the return value is the number of characters copied into the string, not including the terminating null character. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeGetAllHeaders** function will return all of the RFC 822 header values in a string buffer. This includes the message headers that are most commonly referred to, such as the To, From and Subject headers. Each header and its value are separated by a colon, and terminated with a carriage return and linefeed (CRLF) pair.

The headers and their values returned by this function will not be identical to the header block in the original message. If a header value is split across multiple lines, this function will fold the text, returning the complete header value on a single line of text and removing any extraneous whitespace. If the header value has been encoded by the mail client, this function will return the decoded value, not the original encoded value.

Example

```
LPTSTR lpszHeaders = NULL;  
INT nLength;  
  
// Determine the number of characters that should be allocated to store  
// the RFC822 headers  
  
nLength = MimeGetAllHeaders(hMessage, NULL, 0);
```

```
// Allocate the memory for the string buffer that is large enough and
// call MimeGetAllHeaders again

if (nLength > 0)
{
    lpszHeaders = (LPTSTR)LocalAlloc(LPTR, nLength * sizeof(TCHAR));
    if (lpszHeaders == NULL)
        return; // Virtual memory exhausted

    nLength = MimeGetAllHeaders(hMessage, lpszHeaders, nLength);
}

// The lpszHeaders string now contains all of the RFC822 headers for the
// message, with each header terminated by a CRLF sequence

if (lpszHeaders != NULL)
{
    LocalFree((HLOCAL)lpszHeaders);
    lpszHeaders = NULL;
}
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeGetFirstMessageHeader](#), [MimeGetMessageHeader](#), [MimeGetNextMessageHeader](#),
[MimeSetMessageHeader](#)

MimeGetAllRecipients Function

```
INT WINAPI MimeGetAllRecipients(  
    HMESSAGE hMessage,  
    LPCTSTR LpszExtraAddress,  
    LPTSTR LpszRecipients,  
    INT nMaxLength  
);
```

The **MimeGetAllRecipients** function returns a comma-separated list of recipient addresses in a string buffer.

Parameters

hMessage

Handle to the message.

lpszExtraAddress

A pointer to a string which contains one or more additional email addresses that should be included in the list, in addition to those found in the message. If more than one address is specified, each address should be separated by a comma. This parameter may be NULL if there are no extra addresses to include in the recipient list.

lpszRecipients

Pointer to string buffer which will contain a comma-separated list of email addresses when the function returns. This parameter may be NULL, in which case the function will calculate the number of characters needed to store the complete list.

nMaxLength

An integer value which specifies the maximum number of characters that can be stored in the *lpszRecipients* string. If the *lpszRecipients* parameter is NULL, this value must be zero. If the *lpszRecipients* parameter is not NULL, this value must be large enough to store the entire list of addresses.

Return Value

If the function succeeds and the *lpszRecipients* parameter is NULL, the return value is the minimum number of characters that should be allocated to store the list recipient addresses, including the terminating null character. If the *lpszRecipients* parameter is not NULL, then the return value is the number of characters copied into the buffer, not including the terminating null character. If the function returns a value of zero, then the specified message has no recipients. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeGetAllRecipients** function is useful for creating a list of message recipients that can be passed to functions like **SmtplibSendMessage**. If you wish to dynamically allocate the string buffer that will contain the list of recipients, then the *lpszRecipients* parameter should be NULL and the *nMaxLength* parameter should have a value of zero. The function will then return the recommended size of the buffer that should be allocated. This value is guaranteed to be large enough to store the entire list of message recipients, including the terminating null character.

Example

```
LPTSTR lpszRecipients = NULL;
```

```

INT nLength;

// Determine the number of characters that should be allocated to store
// a list of the recipient addresses in the message

nLength = MimeGetAllRecipients(hMessage, NULL, NULL, 0);

// Allocate the memory for the string buffer that is large enough and
// call MimeGetAllRecipients again

if (nLength > 0)
{
    lpszRecipients = (LPTSTR)LocalAlloc(LPTR, nLength * sizeof(TCHAR));
    if (lpRecipients == NULL)
        return; // Virtual memory exhausted

    nLength = MimeGetAllRecipients(hMessage, NULL, lpszRecipients, nLength);
}

// The lpszRecipients string now contains a comma-separated list of
// each recipient address in the message

if (lpszRecipients != NULL)
{
    LocalFree((HLOCAL)lpszRecipients);
    lpszRecipients = NULL;
}

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeEnumMessageRecipients](#), [MimeGetFirstMessageHeader](#), [MimeGetMessageHeader](#),
[MimeGetNextMessageHeader](#), [MimeSetMessageHeader](#)

MimeGetAttachedFileName Function

```
BOOL WINAPI MimeGetAttachedFileName(  
    HMESSAGE hMessage,  
    LPTSTR lpszFileName,  
    INT cchFileName  
);
```

The **MimeGetAttachedFileName** function returns the file name for the attachment to the current message part.

Parameters

hMessage

Handle to the message.

lpszFileName

Pointer to a buffer that will contain the current attachment file name as a string.

cchFileName

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The function will first try to get the filename from the Content-Disposition header field. If this field does not exist, it then attempts to get the name from the Content-Type header field. If neither field exists, the function will fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeAttachFile](#), [MimeEnumAttachments](#), [MimeExtractFile](#), [MimeFindAttachment](#)

MimeGetContentDigest Function

```
INT WINAPI MimeGetContentDigest(  
    HMESSAGE hMessage,  
    LPTSTR LpszDigest,  
    INT cchDigest  
);
```

The **MimeGetContentDigest** function returns an encoded digest of the message.

Parameters

hMessage

Message handle.

lpszDigest

Pointer to a string buffer to contain the MD5 digest for the specified message.

cchDigest

Maximum length of the digest string, in bytes.

Return Value

If the function succeeds, the return value is the length of the message digest string. A value of zero specifies that there is no MD5 digest for the current message. If the function fails, the return value is `MIME_ERROR`. To get extended error information, call **MimeGetLastError**.

Remarks

This function returns the value of the Content-MD5 header field in the main body of the message. If the header exists, it contains the MD5 digest for the message as defined in RFC 1864.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

MimeGetContentLength Function

```
LONG WINAPI MimeGetContentLength(  
    HMESSAGE hMessage  
);
```

The **MimeGetContentLength** function returns the size of the current message part content in bytes.

Parameters

hMessage

Handle to the message.

Return Value

If the function succeeds, the return value is the content length. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

This function will return the size of the content in the current message part as the number of bytes and does not account for any Unicode conversion of text. Exercise caution when using this function to determine the size of the buffer that should be allocated for a function like **MimeGetMessageText**. You should always allocate enough memory to accommodate any potential text conversion and decoding which may occur.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeGetMessageHeader](#), [MimeGetMessagePart](#), [MimeGetMessageText](#), [MimeSetMessagePart](#)

MimeGetErrorString Function

```
INT WINAPI MimeGetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);
```

The **MimeGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **MimeGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeGetLastError](#), [MimeSetLastError](#)

MimeGetExportOptions Function

```
BOOL WINAPI MimeGetExportOptions(  
    HMESSAGE hMessage  
    LPDWORD LpdwOptions  
);
```

The **MimeGetExportOptions** function returns a bitmask that describes current message export options.

Parameters

hMessage

Handle to the message.

lpdwOptions

Pointer to mask of attribute options. The mask is a combination of the following values:

Constant	Description
MIME_OPTION_DEFAULT	The default export options. The headers for the message are written out in a specific consistent order, with custom headers written to the end of the header block regardless of the order in which they were set or imported from another message. If the message contains Bcc, Received, Return-Path, Status or X400-Received header fields, they will not be exported.
MIME_OPTION_KEEORDER	The original order in which the message header fields were set or imported are preserved when the message is exported.
MIME_OPTION_ALLHEADERS	All headers, including the Bcc, Received, Return-Path, Status and X400-Received header fields will be exported. Normally these headers are not exported because they are only used by the mail transport system. This option can be useful when exporting a message to be stored on the local system, but should not be used when exporting a message to be delivered to another user.
MIME_OPTION_NOHEADERS	When exporting a message, the main header block will not be included. This can be useful when creating a multipart message for services which expect MIME formatted data, such as HTTP POST requests. This option should never be used for email messages being submitted using SMTP.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is 0. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeExportMessage](#), [MimeImportMessage](#), [MimeSetExportOptions](#)

MimeGetFileContentType Function

```
UINT WINAPI MimeGetFileContentType(  
    HMESSAGE hMessage,  
    LPCTSTR lpszFileName,  
    LPTSTR lpszContentType,  
    INT cchContentType  
);
```

The **MimeGetFileContentType** function returns the content type for the specified file.

Parameters

hMessage

Handle to the message.

lpszFileName

Pointer to a string which specifies the name of the file for which content type information is returned.

lpszContentType

Pointer to a string buffer that will contain the MIME type for the specified file. This may be a NULL pointer, in which case this parameter is ignored. If a buffer is provided, it is recommended that it be at least 64 characters in length.

cchContentType

An integer which specifies the maximum number of characters, including the terminating null character, which may be copied into the string buffer.

Return Value

If the function succeeds, the return value is the content type of the specified file. If the function fails, the return value is `MIME_CONTENT_UNKNOWN`. To get extended error information, call **MimeGetLastError**.

The following values may be returned by this function:

Constant	Description
<code>MIME_CONTENT_UNKNOWN</code>	The file content type is unknown. This value may be returned if the message handle is invalid, or if the file extension is unknown and the file could not be opened for read access.
<code>MIME_CONTENT_APPLICATION</code>	The file content is application specific. Examples of this type of file would be a Microsoft Word document or an executable program. This is also the default type for files which have an unrecognized file name extension and contain binary data.
<code>MIME_CONTENT_AUDIO</code>	The file is an audio file in one of several standard formats. Examples of this type of file would be a Windows (.wav) file or MPEG3 (.mp3) file.
<code>MIME_CONTENT_IMAGE</code>	The file is an image file in one of several standard formats. Examples of this type of file would be a GIF or JPEG image file.
<code>MIME_CONTENT_TEXT</code>	The file is a text file. This is also the default type for files which have an unrecognized file name extension and contain only

	printable text data.
MIME_CONTENT_VIDEO	The file is a video file in one of several standard formats. Examples of this type of file would be a Windows (.avi) or Quicktime (.mov) video file.

Remarks

The content type for a given file is determined based on the file name extension, or if the extension is not recognized, the actual contents of the file. On 32-bit platforms, the system registry is used to determine the default content type values for a given extension. In all cases, file types that are explicitly set using the **MimeSetFileContentType** function will override the default system values.

The content type string which is copied to the string buffer is the standard MIME content type description, which specifies a primary type and a subtype, separated by a slash. For example, a plain text file would have a content type of text/plain, while an HTML document would have a content type of text/html. Binary files may be associated with a specific application. For example, the content type for a Microsoft Word document is application/msword. Those binary files which are not associated with a specific application, or have an unrecognized file name extension, have a content type of application/octet-stream.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeAttachData](#), [MimeAttachFile](#), [MimeSetFileContentType](#)

MimeGetFirstMessageHeader Function

```
BOOL WINAPI MimeGetFirstMessageHeader(  
    HMESSAGE hMessage,  
    LPTSTR lpszHeader,  
    INT cchMaxHeader,  
    LPTSTR lpszValue,  
    INT cchMaxValue  
);
```

The **MimeGetFirstMessageHeader** function returns the header field name and value for the first header in the current message part. This function is typically used in conjunction with **MimeGetNextMessageHeader** to enumerate all of the message header fields and their values in the current message part.

Parameters

hMessage

Handle to the message.

lpszHeader

A pointer to a string buffer that will contain the name of the first header in the current message part. This parameter cannot be a NULL pointer.

cchMaxHeader

An integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. This parameter must have a value greater than zero.

lpszValue

A pointer to a string buffer that will contain the value of the first header in the current message part. This parameter may be a NULL pointer, in which case the value of the header field is ignored.

cchMaxValue

An integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. If the *lpszValue* parameter is NULL, this parameter should have a value of zero.

Return Value

If the function succeeds, the return value is non-zero. If no headers exist for the current message part, or the handle to the message is invalid, the function will return zero. To get extended error information, call **MimeGetLastError**.

Remarks

Each part in a multipart message has one or more header fields. To obtain header values for the main message, rather than the message attachments, the current part number must be set to zero using the **MimeSetMessagePart** function.

The header fields from an imported message may not be returned in the same order as which they appear in the message. An application should never make an assumption about the order in which one or more header fields are defined.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeEnumMessageHeaders](#), [MimeGetMessagePart](#), [MimeGetNextMessageHeader](#),
[MimeParseHeader](#), [MimeSetMessageHeader](#), [MimeSetMessagePart](#)

MimeGetLastError Function

```
DWORD WINAPI MimeGetLastError();
```

Parameters

None.

Return Value

The return value is the calling thread's last error code value. Functions set this value by calling the **MimeSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last-error code.

Remarks

You should call the **MimeGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **MimeSetLastError(0)** when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_MESSAGE or MIME_ERROR. Those functions which call **MimeSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

See Also

[MimeGetErrorString](#), [MimeSetLastError](#)

MimeGetMessageBoundary Function

```
LPCTSTR WINAPI MimeGetMessageBoundary(  
    HMESSAGE hMessage  
);
```

The **MimeGetMessageBoundary** function returns a pointer to the boundary string used to separate the parts of a multipart message.

Parameters

hMessage

Handle to the message.

Return Value

If the function succeeds, the return value is a pointer to the boundary string. If the function fails, a NULL pointer is returned. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeCreateMessagePart](#), [MimeGetMessageHeader](#), [MimeGetMessagePart](#)

MimeGetMessageDate Function

```
LPCTSTR WINAPI MimeGetMessageDate(  
    HMESSAGE hMessage,  
    BOOL bLocalize  
);
```

The **MimeGetMessageDate** function returns a pointer to a string that contains the message date and time.

Parameters

hMessage

Handle to the message.

bLocalize

Boolean flag which specifies if the date and time should be localized for the current timezone.

Return Value

If the function succeeds, the return value is a pointer to the date and time string. If the function fails, it will return a NULL pointer. To get extended error information, call **MimeGetLastError**.

Remarks

If no date has been specified in the message, the Date header field will be set to the current date and time, and that value will be returned. The date string returned by this function should never be directly modified by the application. Each call to this function will invalidate the previous value that was returned, so if you wish to save or modify the value, you should first make a private copy of the string.

To convert this date string to a long integer value that can be used with the standard C time functions, use the **MimeParseDate** function. Refer to the **MimeFormatDate** function for information on the format of the date string.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeFormatDate](#), [MimeGetMessageHeader](#), [MimeParseDate](#), [MimeSetMessageDate](#)

MimeGetMessageHeader Function

```
LPCTSTR WINAPI MimeGetMessageHeader(  
    HMESSAGE hMessage,  
    LPCTSTR lpszHeader  
);
```

The **MimeGetMessageHeader** function returns a pointer to the value associated with the specified header field in the current message part.

Parameters

hMessage

Handle to the message.

lpszHeader

A pointer to a string which specifies the message header. This parameter cannot be a NULL pointer.

Return Value

If the function succeeds, the return value is a pointer to the header value. If the header field does not exist, a NULL pointer is returned. To get extended error information, call **MimeGetLastError**.

Remarks

Each part in a multipart message has one or more header fields. To obtain header values for the main message, rather than the message attachments, the current part number must be set to zero using the **MimeSetMessagePart** function.

For languages other than C/C++, it may be preferable to use the **MimeGetMessageHeaderEx** function which copies the header value into a string buffer rather than returning a pointer to the header value itself.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeEnumMessageHeaders](#), [MimeGetFirstMessageHeader](#), [MimeGetMessageHeaderEx](#), [MimeGetMessagePart](#), [MimeGetNextMessageHeader](#), [MimeParseHeader](#), [MimeSetMessageHeader](#), [MimeSetMessagePart](#)

MimeGetMessageHeaderEx Function

```
INT WINAPI MimeGetMessageHeaderEx(  
    HMESSAGE hMessage,  
    INT nMessagePart,  
    LPCTSTR lpszHeader,  
    LPTSTR lpszValue,  
    INT nMaxValue,  
    DWORD dwReserved  
);
```

The **MimeGetMessageHeaderEx** copies the value of the specified header into a string buffer.

Parameters

hMessage

Handle to the message.

nMessagePart

An integer value which specifies which part of the message to return the header value from. A value of zero returns a header value from the main message header, while a value greater than zero returns the header value from that specific part of a multipart message. A value of -1 specifies that the header value should be returned from the current message part.

lpszHeader

A pointer to a string which specifies the message header. This parameter cannot be a NULL pointer.

lpszValue

A pointer to a string buffer which will contain the value of the specified header when the function returns. If this parameter is NULL, the function will return the length of the header value without copying the data. This is useful for determining the length of a header value so that a string buffer can be allocated and passed to a subsequent call to the function.

nMaxValue

An integer value which specifies the maximum number of characters that can be copied to the string buffer, including the terminating null character. If the *lpszValue* parameter is NULL, this value should be zero.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is the number of bytes copied into the string buffer, not including the terminating null byte. If the *lpszValue* parameter is NULL, the return value is the length of the header value. If the header field does not exist, a value of zero is returned. If an invalid pointer or message part is specified, a value of MIME_ERROR is returned. To get extended error information, call **MimeGetLastError**.

Example

```
LPTSTR lpszValue = NULL;  
INT cchValue = 0;  
  
// Determine the length of the To header field  
cchValue = MimeGetMessageHeaderEx(hMessage, 0, _T("To"), NULL, 0);
```

```
// If the header field exists, allocate a string buffer
// and copy the value into the buffer
if (cchValue > 0)
{
    lpszValue = (LPTSTR)LocalAlloc(LPTR, cchValue + 1);
    MimeGetMessageHeaderEx(hMessage,
                           0,
                           _T("To"),
                           lpszValue,
                           cchValue + 1);
}

// After the string buffer has been used, release the
// memory that was allocated for it
if (lpszValue)
{
    LocalFree((HLOCAL)lpszValue);
    lpszValue = NULL;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeEnumMessageHeaders](#), [MimeGetFirstMessageHeader](#), [MimeGetMessageHeader](#),
[MimeGetMessagePart](#), [MimeGetNextMessageHeader](#), [MimeParseHeader](#),
[MimeSetMessageHeader](#), [MimeSetMessagePart](#)

MimeGetMessagePart Function

```
INT WINAPI MimeGetMessagePart(  
    HMESSAGE hMessage  
);
```

The **MimeGetMessagePart** function returns the current message part index for the specified message.

Parameters

hMessage

Handle to the message.

Return Value

If the function succeeds, the return value is the message part index. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

See Also

[MimeCreateMessagePart](#), [MimeDeleteMessagePart](#), [MimeGetMessagePartCount](#), [MimeSetMessagePart](#)

MimeGetMessagePartCount Function

```
INT WINAPI MimeGetMessagePartCount(  
    HMESSAGE hMessage  
);
```

The **MimeGetMessagePartCount** function returns the total number of message parts for the specified message. Each message consists of at least one part.

Parameters

hMessage

Handle to the message.

Return Value

If the function succeeds, the return value is the number of message parts. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

See Also

[MimeCreateMessagePart](#), [MimeGetMessagePart](#), [MimeSetMessagePart](#)

MimeGetMessageSender Function

```
INT WINAPI MimeGetMessageSender(  
    HMESSAGE hMessage,  
    LPCTSTR LpszSender,  
    INT nMaxLength  
);
```

The **MimeGetMessageSender** function returns the email address of the message sender in the specified string buffer.

Parameters

hMessage

Handle to the message.

lpszSender

A pointer to a string buffer that will contain the email address of the message sender when the function returns.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied to the string buffer, including the terminating null character. It is recommended that the maximum length be at least 64 characters.

Return Value

If the function succeeds, the return value is the number of bytes copied to the string buffer. If an error occurs, the function will return MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

This function attempts to determine the email address of the sender that originated the message. It will first check for the presence of a Sender or X-Sender header value. If these headers are not defined, it will use the value of the From header field. It will only return successfully if a valid email address can be found.

If the function succeeds, the string buffer that is provided will only contain an email address. It will not contain the display name of the user associated with the address or any extraneous comments that are included in quotes or parenthesis.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeEnumMessageHeaders](#), [MimeGetFirstMessageHeader](#), [MimeGetMessageHeaderEx](#), [MimeGetMessagePart](#), [MimeGetNextMessageHeader](#), [MimeParseHeader](#), [MimeSetMessageHeader](#), [MimeSetMessagePart](#)

MimeGetMessageSize Function

```
BOOL WINAPI MimeGetMessageSize(  
    HMESSAGE hMessage,  
    DWORD dwOptions,  
    LPDWORD lpdwMessageSize  
);
```

The **MimeGetMessageSize** function returns the size of the complete message in bytes.

Parameters

hMessage

Handle to the message.

dwOptions

An unsigned integer which specifies how the size of the message should be calculated, based on what header fields should be included. These are the same options used when exporting a message to a file or memory buffer. The following values may be combined using a bitwise Or operator:

Constant	Description
MIME_OPTION_DEFAULT	The default export options. The headers for the message are written out in a specific consistent order, with custom headers written to the end of the header block regardless of the order in which they were set or imported from another message. If the message contains Bcc, Received, Return-Path, Status or X400-Received header fields, they will not be exported.
MIME_OPTION_ALLHEADERS	All headers, including the Received, Return-Path, Status and X400-Received header fields will be exported. Normally these headers are not exported because they are only used by the mail transport system. This option can be useful when exporting a message to be stored on the local system, but should not be used when exporting a message to be delivered to another user.

lpdwMessageSize

A pointer to an unsigned integer value which will contain the size of the message if the function succeeds, or a value of zero if the function fails. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If an error occurs, the function will return zero. To get extended error information, call **MimeGetLastError**.

Remarks

This function returns the size of the complete message, including all headers, the message body and any attachments. It can be used to determine the minimum amount of memory that should be allocated to export the message to a memory buffer using the **MimeExportMessageEx** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

See Also

[MimeExportMessage](#), [MimeExportMessageEx](#), [MimeGetMessageText](#), [MimeImportMessage](#),
[MimeImportMessageEx](#)

MimeGetMessageText Function

```
LONG WINAPI MimeGetMessageText(  
    HMESSAGE hMessage,  
    LONG nOffset,  
    LPTSTR LpszBuffer,  
    LONG nMaxLength  
);
```

The **MimeGetMessageText** function copies the text of the current message part to the specified buffer.

Parameters

hMessage

Handle to the message.

nOffset

The byte offset from the beginning of the message. A value of zero specifies the first character in the body of the message part.

lpszBuffer

A pointer to the string buffer that will contain a copy of the message text when the function returns. Note that the buffer will typically contain multiple lines of text.

nMaxLength

The maximum number of bytes to copy into the buffer. The size of the buffer provided must be larger than the content length for the current message part.

Return Value

If the function succeeds, the return value is number of bytes copied into the buffer. If the function fails, the return value is `MIME_ERROR`. To get extended error information, call **MimeGetLastError**.

Remarks

If your project targets a multi-byte character set, this function will always return the message contents as UTF-8 text, regardless of the original character set specified in the message itself. This ensures that characters in the original text are preserved, regardless of the default ANSI code page on the local system. It is recommended you build your project to use Unicode whenever possible. If your application must use ANSI, you can use the **MimeLocalizeText** method to convert the Unicode text to a specific character set.

You should not determine the maximum size of the output buffer using the **MimeGetContentLength** function. That function returns the content size in bytes as it is stored in the message, and does not account for any character encoding or Unicode conversion which may be required. The content length can be used to estimate the amount of text stored in the message part, but you should always allocate a buffer which is larger than the length specified in the message.

If the *nMaxLength* parameter does not specify a buffer size large enough to store the contents of the current message part, this function will fail and the last error code will be set to `ST_ERROR_BUFFER_TOO_SMALL`. Your application must ensure the buffer is large enough to contain the complete message text and a terminating NUL character.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeAppendMessageText](#), [MimeClearMessageText](#), [MimeCompareMessageText](#),
[MimeExportMessage](#), [MimeGetContentLength](#), [MimeImportMessage](#), [MimeLocalizeText](#),
[MimeSetMessageText](#)

MimeGetMessageVersion Function

```
LPCTSTR WINAPI MimeGetMessageVersion(  
    HMESSAGE hMessage  
);
```

The **MimeGetMessageVersion** function returns a pointer to a string which contains the MIME version header value for the specified message.

Parameters

hMessage

Handle to the message.

Return Value

If the function succeeds, the return value is a pointer to the MIME version. If the version is not specified in the current message, the function will return NULL. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeGetMessageHeader](#), [MimeSetMessageVersion](#)

MimeGetNextMessageHeader Function

```
BOOL WINAPI MimeGetNextMessageHeader(  
    HMESSAGE hMessage,  
    LPTSTR lpszHeader,  
    INT cchMaxHeader,  
    LPTSTR lpszValue,  
    INT cchMaxValue  
);
```

The **MimeGetNextMessageHeader** function returns the header field name and value for the next header in the current message part. This function is typically used in conjunction with **MimeGetFirstMessageHeader** to enumerate all of the message header fields and their values in the current message part.

Parameters

hMessage

Handle to the message.

lpszHeader

A pointer to the string buffer that will contain the name of the next header in the current message part when the function returns. This parameter cannot be a NULL pointer.

cchMaxHeader

An integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. This parameter must have a value greater than zero.

lpszValue

A pointer to the string buffer that will contain the value of the next header in the current message part when the function returns. This parameter may be a NULL pointer, in which case the value of the header field is ignored.

cchMaxValue

An integer which specifies the maximum number of characters which may be copied into the buffer, including the terminating null character. If the *lpszValue* parameter is NULL, this parameter should have a value of zero.

Return Value

If the function succeeds, the return value is non-zero. If no more headers exist for the current message part, or the handle to the message is invalid, the function will return zero. To get extended error information, call **MimeGetLastError**.

Remarks

Each part in a multipart message has one or more header fields. To obtain header values for the main message, rather than the message attachments, the current part number must be set to zero using the **MimeSetMessagePart** function.

The header fields from an imported message may not be returned in the same order as which they appear in the message. An application should never make an assumption about the order in which one or more header fields are defined, with the following exception:

If an imported message has multiple Received headers, then those headers will be returned by **MimeGetNextMessageHeader** in the order in which they appeared in the original message.

Note that if **MimeGetMessageHeader** is used to retrieve the Received header, the first Received header in the message will be returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeEnumMessageHeaders](#), [MimeGetFirstMessageHeader](#), [MimeGetMessagePart](#),
[MimeParseHeader](#), [MimeSetMessageHeader](#), [MimeSetMessagePart](#)

MimeGetStoredMessage Function

```
HMESSAGE WINAPI MimeGetStoredMessage(  
    HMESSAGESTORE hStorage,  
    LONG nMessageId,  
    DWORD dwOptions  
);
```

The **MimeGetStoredMessage** function retrieves a message from the specified message store.

Parameters

hStorage

Handle to the message store.

nMessageId

An integer value which specifies the message number that should be retrieved. The first message in the message store has a value of one.

dwOptions

A value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
0	A shared message handle returned by the function. The contents of this message will be overwritten each time this function is called.
MIME_COPY_STORED_MESSAGE	A message handle is allocated for a copy of the message that is retrieved from the message store.

Return Value

If the function succeeds, the return value is a handle to the message. If the function fails, the return value is `INVALID_MESSAGE`. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeGetStoredMessage** function returns a message handle for the specified message in the message store. If no options are specified, a temporary message handle is returned that is only valid until the next message is retrieved. The application can use message handle with any function except for the **MimeDeleteMessage** function because it does not own the message handle. If a multithreaded application changes the contents of the temporary message, it will change for all other threads that have obtained a message handle using this function.

If the application must have a unique copy of the message, the `MIME_COPY_STORED_MESSAGE` option should be specified. Instead of returning a handle to a shared message, the message is duplicated and a handle to that copy of the message is returned. If this option is used, the application must call **MimeDeleteMessage** to release the memory allocated for the message.

Example

```
HMESSAGE hMessage = INVALID_MESSAGE;  
LPCTSTR lpszHeader = _T("From");  
LPCTSTR lpszAddress = _T("jsmith@example.com");  
LONG nMessageId = 1;
```

```

// Begin searching for messages from the specified sender
while (nMessageId != MIME_ERROR)
{
    nMessageId = MimeFindStoredMessage(hStorage,
                                       nMessageId,
                                       lpszHeader,
                                       lpszAddress,
                                       MIME_SEARCH_PARTIAL_MATCH);

    if (nMessageId != MIME_ERROR)
    {
        // Get a handle to the message that was found
        hMessage = MimeGetStoredMessage(hStorage, nMessageId, 0);

        if (hMessage != INVALID_MESSAGE)
        {
            // Store the message in a file
            TCHAR szFileName[MAX_PATH];
            BOOL bExported;

            // Create a filename based on the message number
            wsprintf(szFileName, _T("msg%05ld.tmp"), nMessageId);

            // Export the message to a file
            bExported = MimeExportMessage(hMessage, szFileName);
        }

        // Increase the message ID to resume the search at the next message
        nMessageId++;
    }
}

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

See Also

[MimeFindStoredMessage](#), [MimeGetStoredMessageCount](#), [MimeDeleteStoredMessage](#),
[MimeStoreMessage](#)

MimeGetStoredMessageCount Function

```
LONG WINAPI MimeGetStoredMessageCount(  
    HMESSAGESTORE hStorage,  
    LPLONG lpnMessages  
);
```

The **MimeGetStoredMessageCount** function returns the number of messages in the specified message store.

Parameters

hStorage

Handle to the message store.

lpnLastMessage

A pointer to an integer which will contain the message number for the last message in the storage file. If this information is not required, a NULL pointer may be specified.

Return Value

If the function succeeds, the return value is the number of messages in the message store. If the function fails, the return value is MIME_ERROR. To get extended error information, call

MimeGetLastError.

Remarks

The **MimeGetStoredMessageCount** function returns the number of messages in the message store. It is important to note that does not count those messages which have been marked for deletion. This means that the value returned by this function will decrease as messages are deleted.

The message number returned in the *lpnLastMessage* parameter will specify the total number of messages in the message store, including deleted messages.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeFindStoredMessage](#), [MimeGetStoredMessage](#), [MimeDeleteStoredMessage](#)

MimeImportMessage Function

```
BOOL WINAPI MimeImportMessage(  
    HMESSAGE hMessage,  
    LPCTSTR lpszFileName  
);
```

The **MimeImportMessage** function reads the specified file and replaces the current message with its contents.

Parameters

hMessage

Handle to the message.

lpszFileName

Pointer to a string which specifies the name of the text file to import.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

This function will delete the contents of the current message before importing the new message from the specified file.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeAttachFile](#), [MimeExportMessage](#), [MimeExportMessageEx](#), [MimeExtractFile](#), [MimeImportMessageEx](#), [MimeInitialize](#), [MimeParseBuffer](#)

MimeImportMessageEx Function

```
BOOL WINAPI MimeImportMessageEx(  
    HMESSAGE hMessage,  
    DWORD dwImportMode,  
    DWORD dwImportOptions,  
    LPVOID lpvMessage,  
    DWORD dwMessageSize  
);
```

The **MimeImportMessageEx** function imports the contents of a message from a file, string or memory handle.

Parameters

hMessage

Handle to the message.

dwImportMode

An unsigned integer which specifies how the message contents will be imported. It may be one of the following values:

Constant	Description
MIME_IMPORT_DEFAULT	The default import mode. If the <i>lpvMessage</i> parameter is NULL, then the contents of the message will be imported from the system clipboard. Otherwise, the <i>lpvMessage</i> parameter is a pointer to a string which specifies the name of a file that contains the message. The <i>dwMessageSize</i> parameter is ignored.
MIME_IMPORT_FILE	The <i>lpvMessage</i> parameter is a pointer to a string which specifies the name of the file that contains the message. If the file does not exist, or is not a regular text file, an error will occur. The <i>dwMessageSize</i> parameter is ignored.
MIME_IMPORT_CLIPBOARD	The contents of the message is imported from the system clipboard. The <i>lpvMessage</i> parameter is ignored. The <i>dwMessageSize</i> parameter is ignored.
MIME_IMPORT_MEMORY	The contents of the message is imported from a local buffer. The <i>lpvMessage</i> parameter must point to a byte array which contains the message to be imported. The <i>dwMessageSize</i> parameter specifies the number of bytes to copy from the buffer. If this value is zero, it is assumed that the end of the message data in the buffer is terminated with a null character and the length is calculated automatically.
MIME_IMPORT_HGLOBAL	The contents of the message is imported from a global memory buffer. The <i>lpvMessage</i> parameter must be a global memory handle which contains the message. The <i>dwMessageSize</i> parameter specifies the number of bytes to copy from the buffer. If this value is zero, it is assumed

that the end of the message data in the buffer is terminated with a null character and the length is calculated automatically.

dwImportOptions

An unsigned integer which specifies how the message will be imported:

Constant	Description
MIME_OPTION_DEFAULT	The default import options. Currently this is the only valid value for this parameter and applications should always specify this constant.

lpvMessage

A pointer to a string, a byte buffer or a global memory handle. The *dwImportMode* parameter determines how this pointer is used by the function.

dwMessageSize

An unsigned integer value which specifies the size of the message to import. This parameter is only used when importing a message from a memory buffer. The message size is determined automatically when the message is imported from a file or the system clipboard.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Example

The following example imports a message using an HGLOBAL handle that references a block of memory that contains an email message:

```
bResult = MimeImportMessageEx(hMessage,  
                             MIME_IMPORT_HGLOBAL,  
                             MIME_OPTION_DEFAULT,  
                             (LPVOID)hgb1Message,  
                             0);  
  
if (bResult)  
{  
    // The message has been successfully imported  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeCreateMessage](#), [MimeExportMessage](#), [MimeExportMessageEx](#), [MimeImportMessage](#)

MimeInitialize Function

```
BOOL WINAPI MimeInitialize(  
    LPCTSTR lpszLicenseKey,  
    LPINITDATA lpData  
);
```

The **MimeInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpData

Pointer to an INITDATA data structure. This parameter may be NULL if the initialization data for the library is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must insure that this function is called with a valid license key.

If the *lpData* argument is specified, it must point to an INITDATA structure which has been initialized by setting the *dwSize* member to the size of the structure. All other structure members should be set to zero. If the function is successful, then the INITDATA structure will be filled with identifying information about the library.

Although it is only required that **MimeInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **MimeUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **MimeInitialize** or **MimeUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

MimeLocalizeText Function

```
LONG WINAPI MimeLocalizeText(  
    UINT nCharacterSet,  
    LPCTSTR lpszInput,  
    LONG cchInput,  
    LPBYTE lpOutput,  
    LONG cbOutput  
);
```

The **MimeLocalizeText** function converts a Unicode string to its ANSI equivalent using the specified character set.

Parameters

nCharacterSet

A numeric identifier which specifies the [character set](#) to use when localizing the text. A value of zero specifies the locale for the current thread should be used when localizing the text.

lpszInput

A pointer to a null terminated string which contains the Unicode text which should be localized. If the ANSI version of this function is called, the input text must be in UTF-8 format or the function will fail. This parameter cannot be a NULL pointer.

cchInput

An integer value which specifies the number of characters of text in the input string which should be localized. If this value is -1, the entire length of the string up to the terminating null will be decoded.

lpOutput

A pointer to a byte buffer which will contain the localized ANSI version of the input text. This parameter cannot be a NULL pointer.

cbOutput

An integer value which specifies the maximum number of bytes which can be copied into the output buffer. The buffer must be large enough to store all of the localized text. This value must be greater than zero.

Return Value

If the input text can be successfully localized, the return value is the number of bytes copied into the output buffer. If the function returns zero, then no text was localized. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeLocalizeText** function enables the application to localize a Unicode string, returning the ANSI version of that text using the specified character set. Because library handles all text as Unicode internally, the ANSI functions in this library will always return UTF-8 encoded text. This function allows you to easily convert that UTF-8 text to another supported character set.

If the ANSI version of this function is called, the input text must use UTF-8 character encoding or the function will fail with the last error set to ST_ERROR_INVALID_CHARACTER_SET.

If the *nCharacterSet* parameter is MIME_CHARSET_DEFAULT or MIME_CHARSET_UNKNOWN the input text will be converted to the default ANSI code page for the current thread locale. If there are characters in the Unicode input text which cannot be converted to an ANSI equivalent using

the specified character set, those characters will be replaced by the default character for your locale, typically a question mark symbol. You cannot specify `MIME_CHARSET_UTF16` as the character set.

This function will always attempt to ensure that the output buffer is terminated with an extra null byte so it is easier to work with as a standard C-style null terminated string. However, if the output buffer is not large enough to accommodate the extra null byte, it will not be copied. It is always recommended that your output buffer be somewhat larger than the length of the original input text to account for multi-byte character sequences. If the output buffer is not large enough to contain the entire localized text, no bytes will be copied to the output buffer and the function will fail with the last error set to `ST_ERROR_BUFFER_TOO_SMALL`.

This function is only required if your application needs to localize the UTF-8 text returned by another function and convert it to a specific 8-bit ANSI character set. For example, if you have an application which calls the ANSI version of **MimeGetMessageText**, it will return the message contents as UTF-8 text. If you need to display that text as localized ANSI, you can call this function to convert the UTF-8 text to your current locale or a specific character set.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmsgv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeDecodeTextEx](#), [MimeEncodeText](#), [MimeEncodeTextEx](#), [MimeGetMessageText](#),
[MimeSetMessageText](#)

MimeOpenMessageStore Function

```
HMESSAGESTORE WINAPI MimeOpenMessageStore(  
    LPCTSTR lpszFileName,  
    DWORD dwOpenMode  
);
```

The **MimeOpenMessageStore** function opens the specified message storage file.

Parameters

lpszFileName

A pointer to a string which specifies the name of the storage file.

dwOpenMode

A value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
MIME_STORAGE_READ	The message store will be opened for read access. The contents of the message store can be accessed, but cannot be modified by the process unless it has also been opened for writing.
MIME_STORAGE_WRITE	The message store will be opened for writing. This mode also implies read access and must be specified if the application needs to modify the contents of the message store.
MIME_STORAGE_CREATE	The message store will be created if the storage file does not exist. If the file exists, it will be truncated. This mode implies read and write access.
MIME_STORAGE_LOCK	The message store will be opened so that it may only be accessed and modified by the current process.
MIME_STORAGE_COMPRESS	The contents of the message store are compressed. This option is automatically enabled if a compressed message store is opened for reading or writing.
MIME_STORAGE_MAILBOX	The message store should use the UNIX mbox format when reading and storing messages. This option is provided for backwards compatibility and is not recommended for general use.

Return Value

If the function succeeds, the return value is a handle to the message store. If the function fails, the return value is `INVALID_MESSAGESTORE`. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeOpenMessageStore** function opens a message storage file which contains one or more messages. If the storage file is opened for read access, the application can search the message store and extract messages but it cannot add or delete messages. To add new messages

or delete existing messages from the store, it must be opened with write access.

The message store is designed to be a simple, effective way to store multiple messages together in a single file. When the message store is opened, the contents are indexed in memory. Although there is no specific limit to the number of messages that can be stored, there must be sufficient memory available to build an index of each message and its headers. If the application must store and manage a very large number of messages, it is recommended that you use a database rather than a flat-file message store.

Message Store Format

Each message is prefixed by a control sequence of five ASCII 01 characters followed by an ASCII 10 and ASCII 13 character. The messages themselves are stored unmodified in their original text format. The length of each message is calculated based on the location of the control sequence that delimits each message, and explicit message lengths are not stored in the file. This means that it is safe to manually change the message contents, as long as the message delimiters are preserved.

If the message store is compressed, the contents of the storage file are expanded when the file is opened and then re-compressed when the storage file is closed. Using the `MIME_STORAGE_COMPRESS` option reduces the size of the storage file and prevents the contents of the message store from being read using a text file editor. However, enabling compression will increase the amount of memory allocated by the library and can increase the amount of time that it takes to open and close the storage file.

The library also has a backwards compatibility mode where it will recognize storage files that use the UNIX mbox format. While this format is supported for accessing existing files, it is not recommended that you use it when creating new message stores or adding messages to an existing store. There are a number of different variants on the mbox format that have been used by different Mail Transfer Agents (MTAs) on the UNIX platform. For example, the `mboxrd` variant looks identical to the `mboxcl2` variant, and they are programmatically indistinguishable from one another, but they are not compatible. For this reason, the use of the mbox format is strongly discouraged.

Example

```
HMESSAGE hMessage;

// Compose a new message
hMessage = MimeComposeMessage(lpszSender,
                              lpszRecipient,
                              NULL,
                              lpszSubject,
                              lpszMessage,
                              NULL,
                              MIME_CHARSET_DEFAULT,
                              MIME_ENCODING_DEFAULT);

if (hMessage != INVALID_MESSAGE)
{
    HMESSAGESTORE hStorage;

    // Open the message storage file
    hStorage = MimeOpenMessageStore(lpszFileName, MIME_STORAGE_WRITE);

    if (hStorage == INVALID_MESSAGESTORE)
    {
        // Delete the message and return if we are unable to
```

```
        // open the storage file
        MimeDeleteMessage(hMessage);
        return;
    }

    // Store a copy of the message in the message store
    nMessageId = MimeStoreMessage(hStorage, hMessage, 0);

    if (nMessageId == MIME_ERROR)
    {
        // We were unable to store the message
    }

    // Close the message store
    MimeCloseMessageStore(hStorage);

    // Destroy the message that was created
    MimeDeleteMessage(hMessage);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeCloseMessageStore](#), [MimeFindStoredMessage](#), [MimeGetStoredMessage](#),
[MimeGetStoredMessageCount](#), [MimePurgeMessageStore](#)

MimeParseAddress Function

```
INT WINAPI MimeParseAddress(  
    LPCTSTR lpszString,  
    LPCTSTR lpszDomain,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

The **MimeParseAddress** function parses a string for an email address, copying the address to the specified buffer.

Parameters

lpszString

A pointer to a string which contains the email address to parse.

lpszDomain

A pointer to a string which specifies a default domain for the address. This parameter may be NULL or point to an empty string if no default domain is required.

lpszAddress

A pointer to a string buffer which will contain the parsed email address when the function returns. It is recommended that this buffer be at least 128 characters in length.

nMaxLength

The maximum number of characters which can be copied into the string buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the address. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeParseAddress** function is useful for parsing the email addresses that may be specified in various header fields in the message. In many cases, the addresses have additional comment characters which are not part of the address itself. For example, one common format used is as follows:

```
"User Name" <user@domain.com>
```

In this case, the email address is enclosed in angle brackets and the name outside of the brackets is considered to be a comment which is not part of the address itself. Another common format is:

```
user@domain.com (User Name)
```

In this case, there is the address followed by a comment which is enclosed in parenthesis. The **MimeParseAddress** function recognizes both formats, and when passed either string, would return the following address:

```
user@domain.com
```

If there was no domain specified in the address, that is just a user name was specified, then the value the *lpszDomain* parameter is added to the address.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeExportMessage](#), [MimeExtractFile](#), [MimeImportMessage](#)

MimeParseBuffer Function

```
BOOL WINAPI MimeParseBuffer(  
    HMESSAGE hMessage,  
    LPCTSTR lpszBuffer,  
    INT cbBuffer  
);
```

The **MimeParseBuffer** function parses the contents of the specified buffer and adds the contents to the message.

Parameters

hMessage

Handle to the message.

lpszBuffer

Pointer to a buffer that contains the text to be added to the message contents.

cbBuffer

The length of the specified buffer.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

This function is useful when the application needs to parse an arbitrary block of text and add it to the specified message. If the buffer contains header fields, the values will be added to the message header. Once the end of the header block is detected, all subsequent text is added to the body of the message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeImportMessage](#), [MimeImportMessageEx](#), [MimeParseHeader](#)

MimeParseDate Function

```
BOOL WINAPI MimeParseDate(  
    LPCTSTR lpszDate,  
    BOOL bLocalize,  
    LPLONG lpnSeconds,  
    LPLONG lpnTimezone  
);
```

The **MimeParseDate** function parses a message date string, returning the number of seconds since 1 January 1970 and the difference in seconds between the specified timezone and Coordinated Universal Time.

Parameters

lpszDate

A pointer to a string which specifies the date to be parsed. The date string must be in the standard format defined by RFC 822.

bLocalize

A boolean flag which determines if the date should be localized to the current timezone, regardless of the timezone specified in the date string. A non-zero value specifies the timezone for the local system will be used, adjusted for daylight savings time if applicable.

lpnSeconds

A pointer to a long integer which is set to the number of seconds since 1 January 1970 00:00:00 UTC. This date is commonly called the epoch, and is the base date used by the standard C time functions. This pointer may be NULL, in which case the parameter is ignored.

lpnTimezone

A pointer to a long integer which is set to the difference in seconds between the specified date's timezone and coordinated universal time (also known as Greenwich Mean Time). This pointer may be NULL, in which case the parameter is ignored.

Return Value

If the date could be successfully parsed, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

This is not a general purpose date parsing function, and it may not be capable of parsing dates for a specific locale. The date and time should be in a standard format as outlined in RFC 822, which describes the basic structure of Internet email messages. For a description of the date string format, refer to the **MimeFormatDate** function.

If the date and time does not include any timezone information, Coordinated Universal Time (UTC) will be used by default. This is an important consideration if you use this function to parse input from a user, because in most cases they will not provide a timezone and will assume the date and time they enter is for their current timezone.

The value of the *bLocalize* parameter will only change the number of seconds offset by the current timezone and does not affect the value returned in the *lpnSeconds* parameter. If the date and time is localized, the timezone offset will be adjusted for daylight savings if it was in effect at the time.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeFormatDate](#), [MimeGetMessageDate](#), [MimeSetMessageDate](#)

MimeParseHeader Function

```
BOOL WINAPI MimeParseHeader(  
    HMESSAGE hMessage,  
    LPCTSTR LpszBuffer  
);
```

The **MimeParseHeader** function parses a line of text and adds the header and value to the current message.

Parameters

hMessage

Handle to the message.

lpszBuffer

Pointer to a string which contains the header and value to be added to the message.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

This function is used to parse a line of text that is part of a message header. The string must consist of a header name, followed by a colon, followed by the header value. The header name may only consist of printable characters, and may not contain whitespace (space, tab, carriage return or linefeed characters).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeImportMessage](#), [MimeParseBuffer](#)

MimePurgeMessageStore Function

```
BOOL WINAPI MimePurgeMessageStore(  
    HMESSAGESTORE hStorage  
);
```

The **MimePurgeMessageStore** function purges all deleted messages from the specified message store.

Parameters

hStorage

Handle to the message store.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimePurgeMessageStore** function purges all deleted messages from the message store. If the storage file has been opened in read-only mode or there are no messages marked for deletion, this function will take no action.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

See Also

[MimeCopyMessageStore](#), [MimeFindStoredMessage](#), [MimeGetStoredMessage](#),
[MimeDeleteStoredMessage](#)

MimeReplaceStoredMessage Function

```
BOOL WINAPI MimeReplaceStoredMessage(  
    HMESSAGESTORE hStorage,  
    LONG nMessageId,  
    HMESSAGE hMessage,  
    DWORD dwReserved  
);
```

The **MimeReplaceStoredMessage** function replaces the specified message in a message store.

Parameters

hStorage

Handle to the message store.

nMessageId

An integer value which specifies the message number that should be replaced.

hMessage

Handle to the message that will be stored.

dwReserved

A reserved parameter. This value must always be zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeReplaceStoredMessage** function replaces the specified message with a new message. The message number may be a message that has been previously marked for deletion. It is important to note that the change will not be reflected in the physical storage file until it has been closed. If the application needs to replace messages in the message store, it is recommended that the file be opened for exclusive access using the MIME_STORAGE_LOCK option.

Example

```
HMESSAGE hMessage;  
  
// Compose a new message  
hMessage = MimeComposeMessage(lpszSender,  
                               lpszRecipient,  
                               NULL,  
                               lpszSubject,  
                               lpszMessage,  
                               NULL,  
                               MIME_CHARSET_DEFAULT,  
                               MIME_ENCODING_DEFAULT);  
  
if (hMessage != INVALID_MESSAGE)  
{  
    HMESSAGESTORE hStorage;  
    LONG nLastMessage;  
    BOOL bResult;  
  
    // Open the message storage file
```

```

hStorage = MimeOpenMessageStore(lpszFileName, MIME_STORAGE_WRITE);

if (hStorage == INVALID_MESSAGESTORE)
{
    // Delete the message and return if we are unable to
    // open the storage file
    MimeDeleteMessage(hMessage);
    return;
}

if (MimeGetStoredMessageCount(hStorage, &nLastMessage) < 1)
{
    // No messages are stored in the file
    MimeCloseMessageStore(hStorage);
    MimeDeleteMessage(hMessage);
    return;
}

// Replace the last message in the message store
bResult = MimeReplaceStoredMessage(hStorage, nLastMessage, hMessage, 0);

if (bResult == FALSE)
{
    // We were unable to replace the message
}

// Close the message store
MimeCloseMessageStore(hStorage);

// Destroy the message that was created
MimeDeleteMessage(hMessage);
}

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

See Also

[MimeFindStoredMessage](#), [MimeGetStoredMessage](#), [MimeDeleteStoredMessage](#),
[MimeStoreMessage](#)

MimeResetMessage Function

```
BOOL WINAPI MimeResetMessage(  
    HMESSAGE hMessage  
);
```

The **MimeResetMessage** function clears the header and body of the specified message, and deletes all message parts.

Parameters

hMessage

Handle to the message.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

See Also

[MimeCreateMessage](#), [MimeDeleteMessage](#)

MimeSetExportOptions Function

```
BOOL WINAPI MimeSetExportOptions(  
    HMESSAGE hMessage,  
    DWORD dwOptions  
);
```

The **MimeSetExportOptions** function specifies a bitmask that describes current message export options.

Parameters

hMessage

Handle to the message.

dwOptions

Mask of attribute options. The mask is a combination of the following values:

Constant	Description
MIME_OPTION_DEFAULT	The default export options. The headers for the message are written out in a specific consistent order, with custom headers written to the end of the header block regardless of the order in which they were set or imported from another message. If the message contains Bcc, Received, Return-Path, Status or X400-Received header fields, they will not be exported.
MIME_OPTION_KEEPOORDER	The original order in which the message header fields were set or imported are preserved when the message is exported.
MIME_OPTION_ALLHEADERS	All headers, including the Bcc, Received, Return-Path, Status and X400-Received header fields will be exported. Normally these headers are not exported because they are only used by the mail transport system. This option can be useful when exporting a message to be stored on the local system, but should not be used when exporting a message to be delivered to another user.
MIME_OPTION_NOHEADERS	When exporting a message, the main header block will not be included. This can be useful when creating a multipart message for services which expect MIME formatted data, such as HTTP POST requests. This option should never be used for email messages being submitted using SMTP.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

By default, the Received and Return-Path headers are not exported. In addition, the order of the

headers in an exported message is undefined. This is reasonable behavior for most mail clients, but may not be appropriate for applications which need access to all of the header fields.

Example

```
MimeSetExportOptions(hMessage, MIME_EXPORT_OPTIONS_ALL |
                    MIME_EXPORT_OPTIONS_KEEP_ORDER);

if (MimeImportMessage(hMessage, lpszFileName))
{
    // Process the message and make any modifications
    // then write the message back out, preserving all
    // of the headers in their original order
    bResult = MimeExportMessage(hMessage, lpszFileName);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeExportMessage](#), [MimeExportMessageEx](#), [MimeImportMessage](#), [MimeImportMessageEx](#),
[MimeGetExportOptions](#)

MimeSetFileContentType Function

```
BOOL WINAPI MimeSetFileContentType(  
    HMESSAGE hMessage,  
    UINT nContentType,  
    LPCTSTR LpszSubtype,  
    LPCTSTR LpszExtension  
);
```

The **MimeSetFileContentType** function associates a per-message content type with a given file name extension. This association is specific to the message, and is not shared by other messages that may be opened by the process.

Parameters

hMessage

Handle to the message.

nContentType

An identifier which specifies the primary content type. It may be one of the following values:

Constant	Description
MIME_CONTENT_UNKNOWN	The default content type for the specified extension should be used. This value should only be used to delete a previously defined content type.
MIME_CONTENT_APPLICATION	The file content is application specific. Examples of this type of file would be a Microsoft Word document or an executable program. This is also the default type for files which have an unrecognized file name extension and contain binary data.
MIME_CONTENT_AUDIO	The file is an audio file in one of several standard formats. Examples of this type of file would be a Windows (.wav) file or MPEG3 (.mp3) file.
MIME_CONTENT_IMAGE	The file is an image file in one of several standard formats. Examples of this type of file would be a GIF or JPEG image file.
MIME_CONTENT_TEXT	The file is a text file. This is also the default type for files which have an unrecognized file name extension and contain only printable text data.
MIME_CONTENT_VIDEO	The file is a video file in one of several standard formats. Examples of this type of file would be a Windows (.avi) or Quicktime (.mov) video file.

lpszSubtype

A pointer to a string which specifies the MIME subtype. This parameter may be NULL if the content type association is being deleted.

lpszExtension

A pointer to a string which specifies the file name extension that will be associated with the MIME content type.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeSetFileContentType** function allows an application to specify a content type for a given file extension, and is typically used to define custom content types for file attachments. The content type will override any default content types associated with the extension, as well as allow new content types to be defined for application-specific files.

Example

In the following example, the file extension ".dat" is associated with a custom content type, a file is attached to the message and then the custom content type is deleted. Note that because the primary content type designates the file as an application specific (non-text) file, it will be automatically encoded when attached to a message:

```
if (MimeSetFileContentType(hMessage, MIME_CONTENT_APPLICATION, "octet-stream",
"dat"))
{
    bResult = MimeAttachFile(hMessage, lpszFileName, MIME_ATTACH_DEFAULT);
    MimeSetFileContentType(hMessage, MIME_CONTENT_UNKNOWN, NULL, "dat");
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeAttachData](#), [MimeAttachFile](#), [MimeGetFileContentType](#)

MimeSetLastError Function

```
VOID WINAPI MimeSetLastError(  
    DWORD dwErrorCode  
);
```

The **MimeSetLastError** function sets the last error code for the current thread.

Parameters

dwErrorCode

Specifies the last error code for the caller. A value of zero clears the last error code.

Return Value

None.

Remarks

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value error code such as FALSE, NULL, INVALID_MESSAGE or MIME_ERROR. Those functions which call **MimeSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **MimeGetLastError** function. The use of **MimeGetLastError** is optional. An application can call it to find out the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

See Also

[MimeGetErrorString](#), [MimeGetLastError](#)

MimeSetMessageDate Function

```
BOOL WINAPI MimeSetMessageDate(  
    HMESSAGE hMessage,  
    LPCTSTR lpszDate,  
    BOOL bLocalize  
);
```

The **MimeSetMessageDate** function sets the date and time in the header for the specified message.

Parameters

hMessage

Handle to the message.

lpszDate

Pointer to a string which specifies the date and time. If this parameter specifies a zero-length string or a NULL pointer, the current date and time will be used.

bLocalize

Boolean flag that specifies the date and time should be localized for the current timezone.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Remarks

The date string should be in a standard format as outlined in RFC 822, the document which describes the basic structure of Internet email messages. For a description of the date string format, refer to the **MimeFormatDate** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeFormatDate](#), [MimeGetMessageDate](#), [MimeGetMessageHeader](#), [MimeParseDate](#), [MimeSetMessageHeader](#)

MimeSetMessageHeader Function

```
BOOL WINAPI MimeSetMessageHeader(  
    HMESSAGE hMessage,  
    LPCTSTR LpszHeader,  
    LPCTSTR LpszValue  
);
```

The **MimeSetMessageHeader** function adds or updates a header field in the specified message.

Parameters

hMessage

Handle to the message.

lpszHeader

Pointer to a string which specifies the header field that will be added or updated.

lpszValue

Pointer to a string which specifies the value for the header field. This pointer may be NULL, which causes the header field to be removed from the message.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeEnumMessageHeaders](#), [MimeGetFirstMessageHeader](#), [MimeGetMessageHeader](#),
[MimeGetMessagePart](#), [MimeGetNextMessageHeader](#), [MimeSetMessagePart](#),
[MimeSetMessageText](#)

MimeSetMessageHeaderEx Function

```
BOOL WINAPI MimeSetMessageHeaderEx(  
    HMESSAGE hMessage,  
    INT nMessagePart,  
    LPCTSTR lpszHeader,  
    LPCTSTR lpszValue,  
    DWORD dwReserved  
);
```

The **MimeSetMessageHeaderEx** function adds or updates a header field in the specified message.

Parameters

hMessage

Handle to the message.

nMessagePart

An integer value which specifies which part of the message the header should be set or modified in. A value of zero sets a header value in the main message header block, while a value greater than zero sets the header value in that specific part of a multipart message. A value of -1 specifies that the header value should be set in the current message part.

lpszHeader

Pointer to a string which specifies the header field that will be added or updated.

lpszValue

Pointer to a string which specifies the value for the header field. This pointer may be NULL, which causes the header field to be removed from the message.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeEnumMessageHeaders](#), [MimeGetFirstMessageHeader](#), [MimeGetMessageHeader](#), [MimeGetMessagePart](#), [MimeGetNextMessageHeader](#), [MimeSetMessagePart](#), [MimeSetMessageText](#)

MimeSetMessagePart Function

```
INT WINAPI MimeSetMessagePart(  
    HMESSAGE hMessage,  
    INT nNewPart  
);
```

The **MimeSetMessagePart** function sets the current message part index for the specified message.

Parameters

hMessage

Handle to the message.

nNewPart

The new message part index. A value of zero specifies the main message part.

Return Value

If the function succeeds, the return value is the previous message part index. If the function fails, the return value is MIME_ERROR. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeGetMessagePart](#), [MimeGetMessagePartCount](#), [MimeSetMessageHeader](#)

MimeSetMessageText Function

```
LONG WINAPI MimeSetMessageText(  
    HMESSAGE hMessage,  
    LONG nOffset,  
    LPCTSTR lpszText  
);
```

The **MimeSetMessageText** function copies the specified text into the body of the current message part.

Parameters

hMessage

Handle to the message.

nOffset

The offset into the body of the message part. A value of -1 specifies that the text will be appended to the message body.

lpszText

A pointer to a string which specifies the text to be copied to the current message part at the given offset.

Return Value

If the function succeeds, the return value is the number of bytes copied into the message. A return value of zero indicates that no text could be copied into the current message part. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeAppendMessageText](#), [MimeClearMessageText](#), [MimeCompareMessageText](#),
[MimeGetMessageText](#), [MimeResetMessage](#), [MimeSetMessageHeader](#), [MimeSetMessagePart](#)

MimeSetMessageVersion Function

```
BOOL WINAPI MimeSetMessageVersion(  
    HMESSAGE hMessage,  
    LPCTSTR LpszVersion  
);
```

Parameters

hMessage

Handle to the message.

lpszVersion

Pointer to a string which specifies the MIME version. A value of NULL sets the version to the default value of 1.0.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **MimeGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmsgv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[MimeGetMessageVersion](#), [MimeSetMessageHeader](#)

MimeStoreMessage Function

```
LONG WINAPI MimeStoreMessage(  
    HMESSAGESTORE hStorage,  
    HMESSAGE hMessage,  
    DWORD dwReserved  
);
```

The **MimeStoreMessage** function stores the specified message in a message store.

Parameters

hStorage

Handle to the message store.

hMessage

Handle to the message that will be stored.

dwReserved

A reserved parameter. This value must always be zero.

Return Value

If the function succeeds, the return value is the message number for the message that was just stored. If the function fails, the return value is `MIME_ERROR`. To get extended error information, call **MimeGetLastError**.

Remarks

The **MimeStoreMessage** function will always append the specified message to the storage file. If you want to replace a message in the message store, you should use the **MimeReplaceStoredMessage** function.

Example

```
HMESSAGE hMessage;  
  
// Compose a new message  
hMessage = MimeComposeMessage(lpszSender,  
                               lpszRecipient,  
                               NULL,  
                               lpszSubject,  
                               lpszMessage,  
                               NULL,  
                               MIME_CHARSET_DEFAULT,  
                               MIME_ENCODING_DEFAULT);  
  
if (hMessage != INVALID_MESSAGE)  
{  
    HMESSAGESTORE hStorage;  
  
    // Open the message storage file  
    hStorage = MimeOpenMessageStore(lpszFileName, MIME_STORAGE_WRITE);  
  
    if (hStorage == INVALID_MESSAGESTORE)  
    {  
        // Delete the message and return if we are unable to  
        // open the storage file  
        MimeDeleteMessage(hMessage);  
    }  
}
```

```
        return;
    }

    // Store a copy of the message in the message store
    nMessageId = MimeStoreMessage(hStorage, hMessage, 0);

    if (nMessageId == MIME_ERROR)
    {
        // We were unable to store the message
    }

    // Close the message store
    MimeCloseMessageStore(hStorage);

    // Destroy the message that was created
    MimeDeleteMessage(hMessage);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

See Also

[MimeDeleteStoredMessage](#), [MimeFindStoredMessage](#), [MimeGetStoredMessage](#),
[MimeReplaceStoredMessage](#)

MimeUninitialize Function

```
VOID WINAPI MimeUninitialize();
```

The **MimeUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **MimeInitialize** call before it can call any of the other the library functions. When it has completed the use of library, the application must call **MimeUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There must be a call to **MimeUninitialize** for every successful call to **MimeInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmsgv10.lib

See Also

[MimeInitialize](#)

Mail Message Data Structures

- INITDATA

INITDATA Structure

This structure contains information about the SocketTools library when the initialization function returns.

```
typedef struct _INITDATA
{
    DWORD      dwSize;
    DWORD      dwVersionMajor;
    DWORD      dwVersionMinor;
    DWORD      dwVersionBuild;
    DWORD      dwOptions;
    DWORD_PTR  dwReserved1;
    DWORD_PTR  dwReserved2;
    TCHAR      szDescription[128];
} INITDATA, *LPINITDATA;
```

Members

dwSize

Size of this structure. This structure member must be set to the size of the structure prior to calling the initialization function. Failure to do so will cause the function to fail.

dwVersionMajor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the major version number for the library.

dwVersionMinor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the minor version number for the library.

dwVersionBuild

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the build number for the library.

dwOptions

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved1

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved2

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

szDescription

A null terminated string which describes the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

Message Character Sets

Constant	Value	Name	Code Page	Description
MIME_CHARSET_USASCII	1	us-ascii	20127	A character set which defines 7-bit printable characters with values ranging from 20h to 7Eh. An application that uses this character set has the broadest compatibility with most mail servers (MTAs) because it does not require the server to handle 8-bit characters correctly when the message is delivered.
MIME_CHARSET_ISO8859_1	2	iso-8859-1	28591	A character set for most western European languages such as English, French, Spanish and German. This character set is also commonly referred to as Latin-1. This character set is similar to Windows code page 1252 (Windows-1252), however there are differences such as the Euro symbol.
MIME_CHARSET_ISO8859_2	3	iso-8859-2	28592	A character set for most central and eastern European languages such as Czech, Hungarian, Polish and Romanian. This character set is also commonly referred to as Latin-2. This character set is similar to Windows code page 1250, however the characters are arranged differently.
MIME_CHARSET_ISO8859_3	12	iso-8859-3	28593	A character set for southern European languages such as Maltese and Esperanto. This character set was also used with the Turkish language, but it was superseded by ISO 8859-9 which is the preferred character set for Turkish. This character set is not widely used in mail messages and it is recommended that you use UTF-8 instead.
MIME_CHARSET_ISO8859_4	13	iso-8859-4	28594	A character set for northern European languages such as

				Latvian, Lithuanian and Greenlandic. This character set is not widely used in mail messages and it is recommended that you use UTF-8 instead.
MIME_CHARSET_ISO8859_5	4	iso-8859-5	28595	A character set for Cyrillic languages such as Russian, Bulgarian and Serbian. This character set was never widely adopted and most mail messages use either KOI8 or UTF-8 encoding.
MIME_CHARSET_ISO8859_6	5	iso-8859-6	28596	A character set for Arabic languages. Note that the application is responsible for displaying text that uses this character set. In particular, any display engine needs to be able to handle the reverse writing direction and analyze the context of the message to correctly combine the glyphs.
MIME_CHARSET_ISO8859_7	6	iso-8859-7	28597	A character set for the Greek language. This character set is also commonly referred to as Latin/Greek. This character set is no longer widely used and has largely been replaced with UTF-8 which provides more complete coverage of the Greek alphabet.
MIME_CHARSET_ISO8859_8	7	iso-8859-8	28598	A character set for the Hebrew language. Note that similar to Arabic, Hebrew uses a reverse writing direction. An application which displays this character should be capable of processing bi-directional text where a single message may include both right-to-left and left-to-right languages, such as Hebrew and English. In most cases it is recommended that you use UTF-8 instead of this character set.
MIME_CHARSET_ISO8859_9	8	iso-8859-9	28599	A character set for the Turkish language. This character set is

				also commonly referred to as Latin-5. This character set is nearly identical to ISO 8859-1, except that it replaces certain Icelandic characters with Turkish characters.
MIME_CHARSET_ISO8859_10	14	iso-8859-10	28600	A character set for the Danish, Icelandic, Norwegian and Swedish languages. This character set is also commonly referred to as Latin-6 and is similar to ISO 8859-4.
MIME_CHARSET_ISO8859_13	15	iso-8859-13	28603	A character set for Baltic languages. This character set is also commonly referred to as Latin-7. This character set is similar to ISO 8859-4, except it adds certain Polish characters and does not support Nordic languages.
MIME_CHARSET_ISO8859_14	16	iso-8859-14	28604	A character set for Gaelic languages such as Irish, Manx and Scottish Gaelic. This character set is also commonly referred to as Latin-8. This character set replaced ISO 8859-12 which was never fully implemented.
MIME_CHARSET_ISO8859_15	17	iso-8859-15	28605	A character set for western European languages. This character set is also commonly referred to as Latin-9 and is nearly identical to ISO8859-1 except that it replaces lesser-used symbols with the Euro sign and some letters.
MIME_CHARSET_ISO2022_JP	18	iso-2022-jp	50222	A multi-byte character encoding for Japanese that is widely used with mail messages. This is a 7-bit encoding where all characters start with ASCII and uses escape sequences to switch to the double-byte character sets.
MIME_CHARSET_ISO2022_KR	19	iso-2022-kr	50225	A multi-byte character encoding for Korean which encodes both ASCII and Korean

				double-byte characters. This is a 7-bit encoding which uses the shift in and shift out control characters to switch to the double-byte character set.
MIME_CHARSET_ISO2022_CN	20	x-cp50227	50227	A multi-byte character encoding for Simplified Chinese which encodes both ASCII and Chinese double-byte characters. This is a 7-bit encoding which uses the shift in and shift out control characters to switch to the double-byte character set.
MIME_CHARSET_KOI8R	21	koi8-r	20866	A character set for Russian using the Cyrillic alphabet. This character set also covers the Bulgarian language. Most mail messages in the Russian language use this character set or UTF-8 instead of ISO 8859-5, which was never widely adopted.
MIME_CHARSET_KOI8U	22	koi8-u	21866	A character set for Ukrainian using the Cyrillic alphabet. This character set is similar to the KOI8-R character set, but replaces certain symbols with Ukrainian letters. Most mail messages in the Ukrainian language use this character set or UTF-8 instead of ISO 8859-5, which was never widely adopted.
MIME_CHARSET_GB2312	23	x-cp20936	20936	A multi-byte character encoding which can represent ASCII and simplified Chinese characters. It has been superseded by GB18030, however it remains widely used in China.
MIME_CHARSET_GB18030	24	gb18030	54936	A Unicode transformation format which can represent all Unicode code points and supports both simplified and traditional Chinese characters. It is backwards compatible with GB2312 and supersedes that

				character set.
MIME_CHARSET_BIG5	25	big5	950	A multi-byte character set that supports both ASCII characters and traditional Chinese characters. It is widely used in Taiwan, Hong Kong and Macau. It is no longer commonly used in China, which has developed GB18030 as a standard encoding. Microsoft's implementation of Big5 on Windows does not support all of the extensions and is missing certain code points.
MIME_CHARSET_UTF7	9	utf-7	65000	A Unicode transformation format that uses variable-length character encoding to represent Unicode text as a stream of ASCII characters that are safe to transport between mail servers that only support 7-bit printable characters. It is primarily used as an alternative to UTF-8 when quoted-printable or base64 encoding is not desired.
MIME_CHARSET_UTF8	10	utf-8	65001	A Unicode transformation format that uses multi-byte character sequences to represent Unicode text. It is backwards compatible with the ASCII character set, however because it uses 8-bit text, it is recommended that you use either quoted-printable or base64 encoding to ensure compatibility with mail servers that do not support 8-bit characters.
MIME_CHARSET_UTF16	11	utf-16le	N/A	A 16-bit Unicode format that represents each character as a 16-bit value in little endian byte order. This character set is not widely used in mail messages and it is recommended that you use UTF-8 instead. UTF-16 characters in big endian byte order are not supported.

Remarks

When composing a new message, it is recommended that you always use UTF-8 as the character set encoding which ensures broad compatibility with most applications. The other character sets are primarily used when parsing messages generated by other applications. Internally, all message headers and text are processed as UTF-8. If you use the ANSI version of the these functions, header values and message text will always be returned to your application as UTF-8 encoded Unicode, regardless of the original character set used in the message.

In addition to the character sets listed above, the library will recognize additional character sets which correspond to specific Windows code pages, as well several variants. These additional character sets are included for compatibility with other applications; they are not defined because they should not be used when composing new messages.

It is important to note that while certain Windows character sets are similar to standard ISO character sets, they are not identical. For example, although the Windows-1252 character set is nearly identical to ISO 8859-1, they are not interchangeable. Some legacy applications make the error of representing Windows ANSI character sets as 8-bit ISO character sets, which can result in errors when converting them to Unicode. This is something to be aware of when encoding and decoding text generated by older applications. Before the widespread adoption of UTF-8, it was particularly common for legacy Windows mail clients to default to using Windows-1252 for text and label it as using ISO 8859-1.

Although the library supports UTF-16, it is recommended you use UTF-8 instead. Text which uses UTF-16 will always be base64 encoded, and some mail clients may not recognize it as a valid character set. If the message does not specify if big endian or little endian byte order is used, the library will default to little endian. When UTF-16 is used when composing a new message, it will always use little endian byte order.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

See Also

[MimeComposeMessage](#), [MimeCreateMessagePartEx](#), [MimeDecodeTextEx](#), [MimeEncodeTextEx](#)

Network News Transfer Protocol Library

Download and submit articles to a news server.

Reference

- [Functions](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

File Name	CSNWSV10.DLL
Version	10.0.1468.2518
LibID	F37CEE4C-BAFE-43B9-B224-80975DF553B2
Import Library	CSNWSV10.LIB
Dependencies	None
Standards	RFC 977, RFC 2980

Overview

The Network News Transfer Protocol (NNTP) is used with servers that provide news services. This is similar in functionality to bulletin boards or message boards, where topics are organized hierarchically into groups, called newsgroups. Users can browse and search for messages, called news articles, which have been posted by other users. On many servers, they can also post their own articles which can be read by others. The largest collection of public newsgroups available is called USENET, a world-wide distributed discussion system. In addition, there are a large number of smaller news servers. For example, Catalyst Development operates a news server which functions as a forum for technical questions and announcements.

The SocketTools library provides a comprehensive interface for accessing newsgroups, retrieving articles and posting new articles. In combination with the Mail Message library to process the news articles, SocketTools can be used to integrate newsgroup access with an existing email application, or you can implement your own full-featured newsgroup client.

This library supports secure connections using the standard SSL and TLS protocols.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Network News Transfer Protocol Functions

Function	Description
NntpAsyncConnect	Establish an asynchronous connection to the specified server
NntpAttachThread	Attach the specified client handle to another thread
NntpAuthenticate	Authenticate the specified user on the news server
NntpCancel	Cancel the current blocking operation
NntpCloseArticle	Close the article being posted to the current newsgroup
NntpCommand	Send a command to the server
NntpConnect	Connect to the specified server
NntpCreateArticle	Create a new article in the current newsgroup
NntpCreateSecurityCredentials	Create a new security credentials structure
NntpDeleteSecurityCredentials	Delete a previously created security credentials structure
NntpDisableEvents	Disable asynchronous event notification
NntpDisableTrace	Disable logging of socket function calls to the trace log
NntpDisconnect	Disconnect from the current server
NntpEnableEvents	Enable asynchronous event notification
NntpEnableTrace	Enable logging of socket function calls to a file
NntpEventProc	Callback function that processes events generated by the client
NntpFreezeEvents	Suspend asynchronous event processing
NntpGetArticle	Retrieve an article from the server and store the contents in a local buffer
NntpGetArticleEx	Retrieve an article from the server with support for large article IDs
NntpGetArticleByMessageId	Retrieve an article from the server using a message ID and store the contents in a local buffer
NntpGetArticleHeaders	Return the contents of the specified article header
NntpGetArticleHeadersEx	Return the contents of the specified article header with support for large article IDs
NntpGetArticleMessageId	Return the message identifier for the specified article
NntpGetArticleMessageIdEx	Return the message identifier for the specified article with support for large article IDs
NntpGetArticleRange	Return the first and last article number for the current group
NntpGetArticleRangeEx	Return the first and last article number for the current group with support for large article IDs
NntpGetArticleSize	Return the size of the specified news article in bytes
NntpGetArticleSizeEx	Return the size of the specified article with support for large article IDs
NntpGetCurrentArticle	Return the current article number for the selected group
NntpGetCurrentArticleEx	Return the current article number for the selected group with support for large article IDs
NntpGetCurrentDate	Return the current date and time
NntpGetErrorString	Return a description for the specified error code
NntpGetFirstArticle	Return the first available article in the currently selected newsgroup
NntpGetFirstArticleEx	Return the first available article in the currently selected newsgroup with support for large article IDs
NntpGetFirstGroup	Return the first available newsgroup from the server
NntpGetFirstGroupEx	Return the first available newsgroup from the server with support for large article IDs
NntpGetGroupName	Return the name of the currently selected newsgroup
NntpGetGroupTitle	Return a description of the currently selected newsgroup
NntpGetLastError	Return the last error code
NntpGetMessageIdArticle	Return the article number for the specified message identifier

NntpGetMessageIdArticleEx	Return the article number for the specified message with support for large article IDs
NntpGetMultiLine	Return the client multi-line output flag
NntpGetNextArticle	Return the next available article from the current newsgroup
NntpGetNextArticleEx	Return the next available article from the current newsgroup with support for large article IDs
NntpGetNextGroup	Return the next available newsgroup from the server
NntpGetNextGroupEx	Return the next available newsgroup from the server with support for large article IDs
NntpGetResultCode	Return the result code from the previous command
NntpGetResultString	Return the result string from the previous command
NntpGetSecurityInformation	Return security information about the current client connection
NntpGetStatus	Return the current client status
NntpGetTimeout	Return the number of seconds until an operation times out
NntpGetTransferStatus	Return data transfer statistics
NntpGetTransferStatusEx	Return data transfer statistics with support for large article IDs
NntpInitialize	Initialize the library and validate the specified license key at runtime
NntpIsBlocking	Determine if the client is blocked, waiting for information
NntpIsConnected	Determine if the client is connected to the server
NntpIsReadable	Determine if data can be read from the server
NntpIsWritable	Determine if data can be written to the server
NntpListArticles	Return a list of articles in the currently selected newsgroup
NntpListArticlesEx	Return a list of articles in the currently selected newsgroup with support for large article IDs
NntpListGroup	Return a list of newsgroups maintained by the server
NntpListNewGroups	Return a list of newsgroups created since a specified date
NntpOpenArticle	Open the specified article in the current newsgroup
NntpOpenArticleEx	Open the specified article in the current newsgroup with support for large article IDs
NntpOpenArticleByMessageId	Open the specified article by message ID in the current newsgroup
NntpOpenNextArticle	Open the next available article
NntpOpenPreviousArticle	Open the previous article
NntpPostArticle	Post a new article to the news server
NntpRead	Read data returned by the news server
NntpRegisterEvent	Register an event callback function
NntpReset	Reset the client
NntpSelectGroup	Select the specified newsgroup to retrieve articles from
NntpSetLastError	Set the last error code
NntpSetMultiLine	Set the client multi-line output flag
NntpSetTimeout	Set the number of seconds until an operation times out
NntpStoreArticle	Retrieve an article and store the contents in a local file
NntpStoreArticleEx	Retrieve an article and store the contents in a local file with support for large article IDs
NntpUninitialize	Terminate use of the library by the application
NntpWrite	Write data to the news server

NntpAsyncConnect Function

```
HCLIENT WINAPI NntpAsyncConnect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSECURITYCREDENTIALS LpCredentials,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **NntpAsyncConnect** function is used to establish a connection with the server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **NntpConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
NNTP_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
NNTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
NNTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The

	server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
NNTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.
NNTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
NNTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
NNTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpCredentials

Pointer to credentials structure [SECURITYCREDENTIALS](#). This parameter is only used if the `NNTP_OPTION_SECURE` option is specified for the connection. This parameter may be `NULL`, in which case no client credentials will be provided to the server. If client credentials are required, the fields *dwSize*, *lpszCertStore*, and *lpszCertName* must be defined, while other fields may be left undefined. Set *dwSize* to the size of the **SECURITYCREDENTIALS** structure.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than `WM_USER` as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is `INVALID_CLIENT`. To get extended error information, call **NntpGetLastError**.

Remarks

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **NntpAttachThread** function.

Specifying the NNTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
NNTP_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
NNTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
NNTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
NNTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
NNTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
NNTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
NNTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
NNTP_EVENT_LISTARTICLE	The client is retrieving a list of articles from the server and information about another article has been processed. The client

	should use this event to retrieve the next available article from the server.
NNTP_EVENT_LASTARTICLE	The client is retrieving a list of articles from the server and all of the available articles have been processed. This event is generated when there are no further articles.
NNTP_EVENT_LISTGROUP	The client is retrieving a list of newsgroups from the server and information about another group has been processed. The client should use this event to retrieve the next available newsgroup from the server.
NNTP_EVENT_LASTGROUP	The client is retrieving a list of newsgroups from the server and all of the available groups have been processed. This event is generated when there are no further newsgroups.
NNTP_EVENT_PROGRESS	The client is in the process of sending or receiving data from the server. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

To cancel asynchronous notification and return the client to a blocking mode, use the **NntpDisableEvents** function.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpAuthenticate](#), [NntpConnect](#), [NntpCreateSecurityCredentials](#), [NntpDeleteSecurityCredentials](#), [NntpDisconnect](#), [NntpInitialize](#), [NntpUninitialize](#)

NntpAttachThread Function

```
DWORD WINAPI NntpAttachThread(  
    HCLIENT hClient  
    DWORD dwThreadId  
);
```

The **NntpAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **NntpAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **NntpAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **NntpCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **NntpUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csnwsv10.lib

See Also

[NntpCancel](#), [NntpAsyncConnect](#), [NntpConnect](#), [NntpDisconnect](#), [NntpUninitialize](#)

NntpAuthenticate Function

```
INT WINAPI NntpAuthenticate(  
    HCLIENT hClient,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword  
);
```

The **NntpAuthenticate** function is used to authenticate access to the news server. Not all news servers require authentication by the client.

Parameters

hClient

Handle to the client session.

lpszUserName

Pointer to a string which specifies the user name required for authentication on the news server.

lpszPassword

Pointer to a string which specifies the password required for authentication on the news server.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

This function should only be called if the server requires authentication. Two authentication methods, "original" and "simple" authentication, are recognized by the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpAsyncConnect](#), [NntpConnect](#), [NntpListArticles](#), [NntpListGroup](#)s

NntpCancel Function

```
INT WINAPI NntpCancel(  
    HCLIENT hClient  
);
```

The **NntpCancel** function cancels any outstanding blocking operation in the client, causing the blocking function to fail. The application may then retry the operation or terminate the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

When the **NntpCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[NntpIsBlocking](#), [NntpRead](#), [NntpReset](#), [NntpWrite](#)

NntpCloseArticle Function

```
INT WINAPI NntpCloseArticle(  
    HCLIENT hClient  
);
```

The **NntpCloseArticle** function closes the current article that has been opened or created.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

If an article is being created, this function actually submits the article to the server. Note that the client application is responsible for generating the message headers as well as the body of the message. News articles conform to the same general characteristics of an email message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[NntpCreateArticle](#), [NntpOpenArticle](#), [NntpOpenArticleByMessageId](#), [NntpWrite](#)

NntpCommand Function

```
INT WINAPI NntpCommand(  
    HCLIENT hClient,  
    LPCTSTR lpszCommand,  
    LPCTSTR lpszParameter  
);
```

The **NntpCommand** function sends a command to the server and returns the result code back to the caller. This function is typically used for site-specific commands not directly supported by the API.

Parameters

hClient

Handle to the client session.

lpszCommand

The command which will be executed by the server.

lpszParameter

An optional command parameter. If the command requires more than one parameter, then they should be combined into a single string, with a space separating each parameter. If the command does not accept any parameters, this value may be NULL.

Return Value

If the function succeeds, the return value is the result code returned by the server. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

A list of valid commands can be found in the technical specification for the protocol. Many servers will list supported commands when the HELP command is used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetMultiLine](#), [NntpGetResultCode](#), [NntpGetResultString](#), [NntpSetMultiLine](#)

NntpConnect Function

```
HCLIENT WINAPI NntpConnect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSECURITYCREDENTIALS LpCredentials  
);
```

The **NntpConnect** function is used to establish a connection with the server.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
NNTP_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
NNTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
NNTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
NNTP_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.
NNTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the

	use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
NNTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
NNTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpCredentials

Pointer to credentials structure [SECURITYCREDENTIALS](#). This parameter is only used if the `NNTP_OPTION_SECURE` option is specified for the connection. This parameter may be `NULL`, in which case no client credentials will be provided to the server. If client credentials are required, the fields *dwSize*, *lpzCertStore*, and *lpzCertName* must be defined, while other fields may be left undefined. Set *dwSize* to the size of the `SECURITYCREDENTIALS` structure.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is `INVALID_CLIENT`. To get extended error information, call **NntpGetLastError**.

Remarks

To prevent this function from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **NntpConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **NntpAttachThread** function.

Specifying the `NNTP_OPTION_FREETHREAD` option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpAuthenticate](#), [NntpCreateSecurityCredentials](#), [NntpDeleteSecurityCredentials](#),
[NntpDisconnect](#), [NntpInitialize](#), [NntpUninitialize](#)

NntpCreateArticle Function

```
INT WINAPI NntpCreateArticle(  
    HCLIENT hClient  
);
```

The **NntpCreateArticle** function creates a new article in the current newsgroup.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

This function sends the POST command to the news server. Not all servers permit clients to post articles. The client application is responsible for generating the message headers as well as the body of the message. News articles conform to the same general characteristics of an email message.

The **NntpCloseArticle** function must be called once the contents of the article has been written to the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

See Also

[NntpCloseArticle](#), [NntpGetCurrentDate](#), [NntpListArticles](#), [NntpOpenArticle](#),
[NntpOpenArticleByMessageId](#), [NntpOpenNextArticle](#), [NntpOpenPreviousArticle](#), [NntpWrite](#)

NntpCreateSecurityCredentials Function

```
BOOL WINAPI NntpCreateSecurityCredentials(  
    DWORD dwProtocol,  
    DWORD dwOptions,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName,  
    LPVOID lpvReserved,  
    LPSECURITYCREDENTIALS* lppCredentials  
);
```

The **NntpCreateSecurityCredentials** function creates a **SECURITYCREDENTIALS** structure.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is

	supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A pointer to a string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A pointer to a string which specifies the certificate owner's password. A value of NULL specifies

that no password is required. This parameter is only used if a PKCS12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A pointer to a string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

lpvReserved

Pointer reserved for future use. Set it to NULL when using this function.

lppCredentials

Pointer to an [LPSECURITYCREDENTIALS](#) pointer. The memory for the credentials structure will be allocated by this function and must be released by calling the **NntpDeleteSecurityCredentials** function when it is no longer needed. The pointer value must be set to NULL before the function is called. It is important to note that this is a pointer to a pointer variable, not a pointer to the SECURITYCREDENTIALS structure itself.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**.

Remarks

The structure that is created by this function may be used as client credentials when establishing a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can

be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **NntpAsyncConnect** or **NntpConnect** functions.

Example

```
LPSECURITYCREDENTIALS lpSecCred = NULL;
NntpCreateSecurityCredentials(SEcurity_PROTOCOL_DEFAULT,
                             0,
                             NULL,
                             NULL,
                             lpzCertStore,
                             lpzCertName,
                             NULL,
                             &lpSecCred);

hClient = NntpConnect(lpzHostName,
                     NNTP_PORT_SECURE,
                     NNTP_TIMEOUT,
                     NNTP_OPTION_SECURE,
                     lpSecCred);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpAsyncConnect](#), [NntpConnect](#), [NntpDeleteSecurityCredentials](#), [NntpGetSecurityInformation](#), [SECURITYCREDENTIALS](#)

NntpDeleteSecurityCredentials Function

```
VOID WINAPI NntpDeleteSecurityCredentials(  
    LPSECURITYCREDENTIALS* LppCredentials  
);
```

The **NntpDeleteSecurityCredentials** function deletes an existing **SECURITYCREDENTIALS** structure.

Parameters

lppCredentials

Pointer to an **LPSECURITYCREDENTIALS** pointer. On exit from the function, the pointer will be NULL.

Return Value

None.

Example

```
if (lpSecCred)  
    NntpDeleteSecurityCredentials(&lpSecCred);  
  
NntpUninitialize();
```

Remarks

This function can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpCreateSecurityCredentials](#), [NntpUninitialize](#)

NntpDisableEvents Function

```
INT WINAPI NntpDisableEvents(  
    HCLIENT hClient  
);
```

The **NntpDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

This function affects both event notification and event callbacks. Any outstanding events in the message queue should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csnwsv10.lib

See Also

[NntpEnableEvents](#), [NntpFreezeEvents](#), [NntpRegisterEvent](#)

NntpDisableTrace Function

```
BOOL WINAPI NntpDisableTrace();
```

The **NntpDisableTrace** function disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csntsv10.lib

See Also

[NntpEnableTrace](#)

NntpDisconnect Function

```
INT WINAPI NntpDisconnect(  
    HCLIENT hClient  
);
```

The **NntpDisconnect** function terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

See Also

[NntpAsyncConnect](#), [NntpConnect](#), [NntpUninitialize](#)

NntpEnableEvents Function

```
INT WINAPI NntpEnableEvents(  
    HCLIENT hClient,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **NntpEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **NntpRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the event notification window. This window receives a user-defined message which specifies the event that has occurred. If this value is NULL, event notification is disabled.

uEventMsg

An unsigned integer which specifies the user-defined message that is sent when an event occurs. This parameter's value must be greater than the value of WM_USER. If the *hEventWnd* parameter is NULL, this value must be specified as WM_NULL.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
NNTP_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
NNTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
NNTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.

NNTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
NNTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
NNTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
NNTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
NNTP_EVENT_LISTARTICLE	The client is retrieving a list of articles from the server and information about another article has been processed. The client should use this event to retrieve the next available article from the server.
NNTP_EVENT_LASTARTICLE	The client is retrieving a list of articles from the server and all of the available articles have been processed. This event is generated when there are no further articles.
NNTP_EVENT_LISTGROUP	The client is retrieving a list of newsgroups from the server and information about another group has been processed. The client should use this event to retrieve the next available newsgroup from the server.
NNTP_EVENT_LASTGROUP	The client is retrieving a list of newsgroups from the server and all of the available groups have been processed. This event is generated when there are no further newsgroups.
NNTP_EVENT_PROGRESS	The client is in the process of sending or receiving data from the server. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

As noted, some events are only generated when the client is asynchronous mode. These events depend on the Windows Sockets asynchronous notification mechanism.

If event notification is disabled by specifying a NULL window handle, there may still be outstanding events in the message queue that must be processed. Since event handling has been disabled, these events should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[NntpDisableEvents](#), [NntpFreezeEvents](#), [NntpRegisterEvent](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

NntpEnableTrace Function

```
BOOL WINAPI NntpEnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **NntpEnableTrace** function enables the logging of Windows Sockets function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace function logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpDisableTrace](#)

NntpEventProc Function

```
VOID CALLBACK NntpEventProc(  
    HCLIENT hClient,  
    UINT nEvent,  
    DWORD dwError,  
    DWORD_PTR dwParam  
);
```

The **NntpEventProc** function is an application-defined callback function that processes events generated by the client.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **NntpRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **NntpRegisterEvent** function. The **NntpEventProc** function is a placeholder for the application-defined function name.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpDisableEvents](#), [NntpEnableEvents](#), [NntpFreezeEvents](#), [NntpRegisterEvent](#)

NntpFreezeEvents Function

```
INT WINAPI NntpFreezeEvents(  
    HCLIENT hClient,  
    BOOL bFreeze  
);
```

The **NntpFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[NntpDisableEvents](#), [NntpEnableEvents](#), [NntpRegisterEvent](#)

NntpGetArticle Function

```
INT WINAPI NntpGetArticle(  
    HCLIENT hClient,  
    LONG nArticleId,  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwReserved  
);
```

The **NntpGetArticle** function retrieves the specified article and copies the contents to a local buffer.

Parameters

hClient

Handle to the client session.

nArticleId

Number of article to retrieve from the server. This value must be greater than zero.

lpvBuffer

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvBuffer* parameter. If the *lpvBuffer* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the file that was downloaded.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetArticle** function is used to retrieve an article from the server and copy it into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the article. In this case, the *lpvBuffer* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvBuffer* parameter point to a global memory handle which will contain the article data when the function returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

This function will cause the current thread to block until the complete article has been retrieved, a timeout occurs or the operation is canceled. During the transfer, the NNTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls.

Event notification must be enabled, either by calling **NntpEnableEvents**, or by registering a callback function using the **NntpRegisterEvent** function.

To determine the current status of a transfer while it is in progress, use the **NntpGetTransferStatus** function. If you need to retrieve an article based on its message ID rather than the article number, use the **NntpGetArticleByMessageId** function.

Your application should use the **NntpGetArticleEx** function if the server uses 64-bit article IDs.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Return the article into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter
nResult = NntpGetArticle(hClient,
                        nArticleId,
                        &hgblBuffer,
                        &cbBuffer,
                        0);

if (nResult != NNTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // article text
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[NntpCreateArticle](#), [NntpEnableEvents](#), [NntpGetArticleEx](#), [NntpGetArticleByMessageId](#), [NntpGetArticleHeaders](#), [NntpGetTransferStatus](#), [NntpRegisterEvent](#)

NntpGetArticleByMessageId Function

```
INT WINAPI NntpGetArticleByMessageId(  
    HCLIENT hClient,  
    LPCTSTR lpszMessageId,  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwReserved  
);
```

The **NntpGetArticleByMessageId** function retrieves an article using the specified message ID and copies the contents to a local buffer.

Parameters

hClient

Handle to the client session.

lpszMessageId

A pointer to a string which specifies the message ID of the article that you want to retrieve. If this parameter is a NULL pointer or specifies a zero-length string, an error will be returned.

lpvBuffer

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvBuffer* parameter. If the *lpvBuffer* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the file that was downloaded.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetArticleByMessageId** function is used to retrieve an article from the server and copy it into a local buffer. This function is identical to the **NntpGetArticle** function except that a message ID string is used to identify the article, rather than an article number. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the article. In this case, the *lpvBuffer* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvBuffer* parameter point to a global memory handle which will contain the article data when the function returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

This function will cause the current thread to block until the complete article has been retrieved, a timeout occurs or the operation is canceled. During the transfer, the NNTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **NntpEnableEvents**, or by registering a callback function using the **NntpRegisterEvent** function.

To determine the current status of a transfer while it is in progress, use the **NntpGetTransferStatus** function.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Return the article into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter
nResult = NntpGetArticleById(hClient,
                             lpzMessageId,
                             &hgblBuffer,
                             &cbBuffer,
                             0);

if (nResult != NNTP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // article text
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpCreateArticle](#), [NntpEnableEvents](#), [NntpGetArticle](#), [NntpGetArticleHeaders](#),
[NntpGetTransferStatus](#), [NntpRegisterEvent](#)

NntpGetArticleHeaders Function

```
INT WINAPI NntpGetArticleHeaders(  
    HCLIENT hClient,  
    LONG nArticleId,  
    LPVOID lpvHeaders,  
    LPDWORD lpdwLength  
);
```

The **NntpGetArticleHeaders** function retrieves the headers for the specified article from the server.

Parameters

hClient

Handle to the client session.

nArticleId

Number of article to retrieve from the server. This value must be greater than zero.

lpvHeaders

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvHeaders* parameter. If the *lpvHeaders* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the message that was downloaded.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetArticleHeaders** function is used to retrieve an article header block from the server and copy it into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the *lpvHeaders* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvHeaders* parameter point to a global memory handle which will contain the message headers when the function returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur.

This function will cause the current thread to block until the transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the NNTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **NntpEnableEvents**, or by registering a callback function using the **NntpRegisterEvent** function.

Your application should use the **NntpGetArticleHeadersEx** function if the server uses 64-bit article IDs.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[NntpGetArticle](#), [NntpGetArticleHeadersEx](#), [NntpGetArticleRange](#), [NntpListArticles](#), [NntpPostArticle](#)

NntpGetArticleHeadersEx Function

```
INT WINAPI NntpGetArticleHeadersEx(  
    HCLIENT hClient,  
    ULONGLONG nArticleId,  
    LPVOID lpvHeaders,  
    LPDWORD lpdwLength  
);
```

The **NntpGetArticleHeadersEx** function retrieves the headers for the specified article from the server.

Parameters

hClient

Handle to the client session.

nArticleId

Number of article to retrieve from the server.

lpvHeaders

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvHeaders* parameter. If the *lpvHeaders* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the message that was downloaded.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetArticleHeadersEx** function is used to retrieve an article header block from the server and copy it into a local buffer. This function provides support for servers which use 64-bit article IDs. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the *lpvHeaders* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvHeaders* parameter point to a global memory handle which will contain the message headers when the function returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur.

This function will cause the current thread to block until the transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the NNTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **NntpEnableEvents**, or by registering a callback function using the **NntpRegisterEvent** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[NntpGetArticle](#), [NntpGetArticleRangeEx](#), [NntpListArticlesEx](#), [NntpPostArticle](#)

NntpGetArticleMessageId Function

```
INT WINAPI NntpGetArticleMessageId(  
    HCLIENT hClient,  
    LONG nArticleId,  
    LPTSTR lpszMessageId,  
    INT cbMessageId  
);
```

The **NntpGetArticleMessageId** function returns the message identifier for the specified article in the current newsgroup.

Parameters

hClient

Handle to the client session.

nArticleId

Article number to retrieve the message identifier for. The value may be zero, in which case the current article number is used.

lpszMessageId

Pointer to a string buffer which will contain the message identifier for the specified article.

cbMessageId

Maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the message identifier string. If the function fails, the return value is NNTP_ERROR. To get extended error information, call

NntpGetLastError.

Remarks

The message identifier is a string which can uniquely identify the message on the news server. This value may be used to retrieve the contents of the article. Your application should use the

NntpGetArticleMessageIdEx function if the server uses 64-bit article IDs.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csnews10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetArticleMessageIdEx](#), [NntpGetMessageIdArticle](#), [NntpListArticles](#)

NntpGetArticleMessageIdEx Function

```
INT WINAPI NntpGetArticleMessageIdEx(  
    HCLIENT hClient,  
    ULONGLONG nArticleId,  
    LPTSTR lpszMessageId,  
    INT cbMessageId  
);
```

The **NntpGetArticleMessageIdEx** function returns the message identifier for the specified article in the current newsgroup.

Parameters

hClient

Handle to the client session.

nArticleId

Article number to retrieve the message identifier for. The value may be zero, in which case the current article number is used.

lpszMessageId

Pointer to a string buffer which will contain the message identifier for the specified article.

cbMessageId

Maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the message identifier string. If the function fails, the return value is NNTP_ERROR. To get extended error information, call

NntpGetLastError.

Remarks

The message identifier is a string which can uniquely identify the message on the news server. This value may be used to retrieve the contents of the article. This function provides support for servers which use 64-bit article IDs.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csnews10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetMessageIdArticleEx](#), [NntpListArticlesEx](#)

NntpGetArticleRange Function

```
LONG WINAPI NntpGetArticleRange(  
    HCLIENT hClient,  
    LPLONG lpnFirstArticle,  
    LPLONG lpnLastArticle  
);
```

The **NntpGetArticleRange** function returns the first and last article numbers for the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

lpnFirstArticle

Pointer to a long integer that will contain the first article number in the currently selected newsgroup. If this parameter is NULL, it will be ignored.

lpnLastArticle

Pointer to a long integer that will contain the last article number in the currently selected newsgroup. If this parameter is NULL, it will be ignored.

Return Value

If the function succeeds, the return value is number of articles in the selected newsgroup. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

It is possible that there will be gaps in the articles within the range of the first and last articles in the newsgroup. This may be due to a message being canceled or expired. If the server uses 64-bit article IDs, your application should use the **NntpGetArticleRangeEx** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csnewsv10.lib

See Also

[NntpGetArticleRangeEx](#), [NntpOpenArticle](#), [NntpOpenArticleByMessageId](#), [NntpListArticles](#)

NntpGetArticleRangeEx Function

```
BOOL WINAPI NntpGetArticleRangeEx(  
    HCLIENT hClient,  
    ULONGLONG * LpnFirstArticle,  
    ULONGLONG * LpnLastArticle,  
    ULONGLONG * LpnArticleCount  
);
```

The **NntpGetArticleRangeEx** function returns the first and last article numbers for the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

lpnFirstArticle

Pointer to an unsigned 64-bit integer that will contain the first article number in the currently selected newsgroup. If this parameter is NULL, it will be ignored.

lpnLastArticle

Pointer to an unsigned 64-bit integer that will contain the last article number in the currently selected newsgroup. If this parameter is NULL, it will be ignored.

lpnArticleCount

Pointer to an unsigned 64-bit integer that will contain the total number of articles in the selected newsgroup. If this parameter is NULL, it will be ignored.

Return Value

If the function succeeds, the return value will be non-zero. If the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**.

Remarks

It is possible that there will be gaps in the articles within the range of the first and last articles in the newsgroup. This may be due to a message being canceled or expired. This function provides support for servers which use 64-bit article IDs.

The **NntpGetArticleRangeEx** function returns a BOOL (a signed 32-bit integer) to indicate success or failure, and the article count is returned in a variable that is passed by reference to the function. If you are updating your code to use this function, make sure you also change how you check the return value.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

See Also

[NntpOpenArticleEx](#), [NntpOpenArticleByMessageId](#), [NntpListArticlesEx](#)

NntpGetArticleSize Function

```
DWORD WINAPI NntpGetArticleSize(  
    HCLIENT hClient,  
    LONG nArticleId  
);
```

The **NntpGetArticleSize** function returns the size of the specified article.

Parameters

hClient

Handle to the client session.

nArticleId

An integer value that identifies the article. The value may be zero, in which case the current article number is used.

Return Value

If the function succeeds, the return value is the size of the article in bytes. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetArticleSize** function sends the XOVER command to the server to request the size of the specified article. If the server uses 64-bit article IDs, your application should use the **NntpGetArticleSizeEx** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[NntpGetArticleSizeEx](#), [NntpOpenArticle](#), [NntpOpenArticleByMessageId](#), [NntpListArticles](#)

NntpGetArticleSizeEx Function

```
DWORD WINAPI NntpGetArticleSizeEx(  
    HCLIENT hClient,  
    ULONGLONG nArticleId  
);
```

The **NntpGetArticleSize** function returns the size of the specified article.

Parameters

hClient

Handle to the client session.

nArticleId

An unsigned 64-bit integer value that identifies the article. The value may be zero, in which case the current article number is used.

Return Value

If the function succeeds, the return value is the size of the article in bytes. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetArticleSizeEx** function sends the XOVER command to the server to request the size of the specified article. This function provides support for servers which use 64-bit article IDs.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

See Also

[NntpOpenArticleEx](#), [NntpOpenArticleByMessageId](#), [NntpListArticlesEx](#)

NntpGetCurrentArticle Function

```
LONG WINAPI NntpGetCurrentArticle(  
    HCLIENT hClient  
);
```

The **NntpGetCurrentArticle** function returns the current article number for the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the current article number. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpOpenArticle](#), [NntpOpenArticleByMessageId](#), [NntpGetArticleRange](#), [NntpGetFirstArticle](#), [NntpGetNextArticle](#), [NntpListArticles](#)

NntpGetCurrentArticleEx Function

```
BOOL WINAPI NntpGetCurrentArticleEx(  
    HCLIENT hClient,  
    ULONGLONG * lpnArticleId  
);
```

The **NntpGetCurrentArticle** function returns the current article number for the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

lpnArticleId

Pointer to an unsigned 64-bit integer that will contain the current article number in the selected newsgroup. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetCurrentArticleEx** function returns a BOOL (a signed 32-bit integer) to indicate success or failure, and the article ID is returned in a variable that is passed by reference to the function. This function provides support for servers which use 64-bit article IDs. If you are updating your code to use this function, make sure you also change how you check the return value.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpOpenArticleEx](#), [NntpOpenArticleByMessageId](#), [NntpGetArticleRangeEx](#), [NntpGetFirstArticleEx](#), [NntpGetNextArticleEx](#), [NntpListArticlesEx](#)

NntpGetCurrentDate Function

```
INT WINAPI NntpGetCurrentDate(  
    LPTSTR lpszDate,  
    INT nMaxLength  
);
```

The **NntpGetCurrentDate** function copies the current date and time to the specified buffer in a format that is commonly used in news articles. This date format should be used in all date-related fields in the message header.

Parameters

lpszDate

Pointer to a string buffer that will contain the current date and time when the function returns.

nMaxLength

The maximum number of characters that can be copied into the string buffer.

Return Values

If the function succeeds, the return value is the number of characters copied into the buffer, not including the null-terminator. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The date value that is returned is adjusted for the local timezone.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpCreateArticle](#)

NntpGetErrorString Function

```
INT WINAPI NntpGetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);
```

The **NntpGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **NntpGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetLastError](#), [NntpSetLastError](#)

NntpGetFirstArticle Function

```
BOOL WINAPI NntpGetFirstArticle(  
    HCLIENT hClient,  
    LPNEWSARTICLE lpArticle  
);
```

The **NntpGetFirstArticle** function returns information about the first article in the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

lpArticle

A pointer to a [NEWSARTICLE](#) structure which will contain information about the first article in the currently selected directory.

Return Value

If the function succeeds, the return value is non-zero. If there are no articles in the current newsgroup, or the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetFirstArticle** function returns information about the first article in the currently selected newsgroup. This function is used in conjunction with the **NntpGetNextArticle** function to enumerate all of the articles in the newsgroup. Typically this is used to provide the user with a list of articles to access.

While the articles in the newsgroup are being listed, the client cannot retrieve the contents of a specific article. For example, the **NntpGetArticle** function cannot be called while inside a loop calling **NntpGetNextArticle**. The client should store those articles which it wants to retrieve in an array, and then once all of the articles have been listed, it can begin calling **NntpGetArticle** for each article number to retrieve the article text.

The date and time that the article was posted is returned in the *stPosted* member of the [NEWSARTICLE](#) structure. This value is returned in Universal Coordinated Time (UTC) and can be converted to local time using the **SystemTimeToTzSpecificLocalTime** function.

If the server uses 64-bit article IDs, your application should use the **NntpGetFirstArticleEx** and **NntpGetNextArticleEx** functions.

Example

```
NEWSARTICLE newsArticle;  
BOOL bResult;  
INT nResult;  
  
// List all articles in the current group  
nResult = NntpListArticles(hClient, -1L, -1L);  
  
if (nResult == NNTP_ERROR)  
{  
    TCHAR szError[ASTRING];  
    DWORD dwError = NntpGetLastError();
```

```
NntpGetErrorString(dwError, szError, ASTRING);
_ftprintf(stderr, _T("Error %08x: %s\n"), dwError, szError);
return;
}

// Get each article in the current newsgroup, printing the article
// number and the subject of the article
bResult = NntpGetFirstArticle(hClient, &newsArticle);

while (bResult)
{
    _tprintf(_T("%ld %s\n"), newsArticle.nArticleId, newsArticle.szSubject);
    bResult = NntpGetNextArticle(hClient, &newsArticle);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetArticle](#), [NntpGetFirstArticleEx](#), [NntpGetFirstGroup](#), [NntpGetNextArticle](#),
[NntpGetNextGroup](#), [NntpGetNextGroupEx](#), [NntpListArticles](#), [NntpListGroups](#), [NntpListNewGroups](#),
[NntpSelectGroup](#), [NEWSARTICLE](#), [NEWSGROUP](#)

NntpGetFirstArticleEx Function

```
BOOL WINAPI NntpGetFirstArticleEx(  
    HCLIENT hClient,  
    LPNEWSARTICLEEX lpArticleEx  
);
```

The **NntpGetFirstArticleEx** function returns information about the first article in the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

lpArticleEx

A pointer to a [NEWSARTICLEEX](#) structure which will contain information about the first article in the currently selected directory.

Return Value

If the function succeeds, the return value is non-zero. If there are no articles in the current newsgroup, or the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetFirstArticleEx** function returns information about the first article in the currently selected newsgroup. This function is used in conjunction with the **NntpGetNextArticleEx** function to enumerate all of the articles in the newsgroup. Typically this is used to provide the user with a list of articles to access. This function provides support for servers which use 64-bit article IDs.

While the articles in the newsgroup are being listed, the client cannot retrieve the contents of a specific article. For example, the **NntpGetArticleEx** function cannot be called while inside a loop calling **NntpGetNextArticleEx**. The client should store those articles which it wants to retrieve in an array, and then once all of the articles have been listed, it can begin calling **NntpGetArticleEx** for each article number to retrieve the article text.

The date and time that the article was posted is returned in the *stPosted* member of the NEWSARTICLE structure. This value is returned in Universal Coordinated Time (UTC) and can be converted to local time using the **SystemTimeToTzSpecificLocalTime** function.

Example

```
NEWSARTICLEEX newsArticle;  
BOOL bResult;  
INT nResult;  
  
// List all articles in the current group  
nResult = NntpListArticlesEx(hClient, (ULONGLONG)-1, (ULONGLONG)-1);  
  
if (nResult == NNTP_ERROR)  
{  
    TCHAR szError[ASTRING];  
    DWORD dwError = NntpGetLastError();  
  
    NntpGetErrorString(dwError, szError, ASTRING);  
    _ftprintf(stderr, _T("Error %08x: %s\n"), dwError, szError);  
}
```

```
    return;
}

// Get each article in the current newsgroup, printing the article
// number and the subject of the article
bResult = NntpGetFirstArticleEx(hClient, &newsArticle);

while (bResult)
{
    _tprintf(_T("%I64u %s\n"), newsArticle.nArticleId, newsArticle.szSubject);
    bResult = NntpGetNextArticleEx(hClient, &newsArticle);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetArticleEx](#), [NntpGetFirstGroupEx](#), [NntpGetNextArticleEx](#), [NntpGetNextGroupEx](#),
[NntpListArticlesEx](#), [NntpListGroupsEx](#), [NntpListNewGroups](#), [NntpSelectGroup](#), [NEWSARTICLEEX](#),
[NEWSGROUPEX](#)

NntpGetFirstGroup Function

```
BOOL WINAPI NntpGetFirstGroup(  
    HCLIENT hClient,  
    LPNEWSGROUP lpGroup  
);
```

The **NntpGetFirstGroup** function returns information about the first available newsgroup.

Parameters

hClient

Handle to the client session.

lpGroup

A pointer to a [NEWSGROUP](#) structure which will contain information about the first available newsgroup.

Return Value

If the function succeeds, the return value is non-zero. If there are no newsgroups available, or the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetFirstGroup** function returns information about the first newsgroup on the server. This function is used in conjunction with the **NntpGetNextGroup** function to enumerate all of the available newsgroups. Typically this is used to provide the user with a list of newsgroups to select.

While the the newsgroups are being listed, the client cannot select a newsgroup or retrieve the contents of a specific article. The client should store those newsgroups which it wants to retrieve articles from, and then once all of the newsgroups have been listed, it can then select each newsgroup and retrieve the available articles from that group.

Note that if no newsgroups are returned by the server, it may indicate that it requires the client to authenticate itself prior to requesting a list of groups or articles.

If the server uses 64-bit article IDs, your application should use the **NntpGetFirstGroupEx** and **NntpGetNextGroupEx** functions.

Example

```
NEWSGROUP newsGroup;  
BOOL bResult;  
INT nResult;  
  
// List all available newsgroups  
nResult = NntpListGroup(hClient);  
  
if (nResult == NNTP_ERROR)  
{  
    TCHAR szError[ASTRING];  
    DWORD dwError = NntpGetLastError();  
  
    NntpGetErrorString(dwError, szError, ASTRING);  
    _ftprintf(stderr, _T("Error %08x: %s\n"), dwError, szError);  
    return;  
}
```

```
// Get each newsgroup, printing the article range and
// the name of the group
bResult = NntpGetFirstGroup(hClient, &newsGroup);

while (bResult)
{
    _tprintf(_T("%ld %ld %s\n"), newsGroup.nFirstArticle,
            newsGroup.nLastArticle,
            newsGroup.szName);

    bResult = NntpGetNextGroup(hClient, &newsGroup);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetFirstArticle](#), [NntpGetFirstGroupEx](#), [NntpGetNextArticle](#), [NntpGetNextGroup](#),
[NntpGetNextGroupEx](#), [NntpListArticles](#), [NntpListGroups](#), [NntpListNewGroups](#), [NntpSelectGroup](#),
[NEWSARTICLE](#), [NEWSGROUP](#)

NntpGetFirstGroupEx Function

```
BOOL WINAPI NntpGetFirstGroupEx(  
    HCLIENT hClient,  
    LPNEWSGROUPEX lpGroupEx  
);
```

The **NntpGetFirstGroupEx** function returns information about the first available newsgroup.

Parameters

hClient

Handle to the client session.

lpGroupEx

A pointer to a [NEWSGROUPEX](#) structure which will contain information about the first available newsgroup.

Return Value

If the function succeeds, the return value is non-zero. If there are no newsgroups available, or the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetFirstGroupEx** function returns information about the first newsgroup on the server. This function is used in conjunction with the **NntpGetNextGroupEx** function to enumerate all of the available newsgroups. Typically this is used to provide the user with a list of newsgroups to select. This function provides support for servers which use 64-bit article IDs.

While the the newsgroups are being listed, the client cannot select a newsgroup or retrieve the contents of a specific article. The client should store those newsgroups which it wants to retrieve articles from, and then once all of the newsgroups have been listed, it can then select each newsgroup and retrieve the available articles from that group.

Note that if no newsgroups are returned by the server, it may indicate that it requires the client to authenticate itself prior to requesting a list of groups or articles.

Example

```
NEWSGROUPEX newsGroup;  
BOOL bResult;  
INT nResult;  
  
// List all available newsgroups  
nResult = NntpListGroupEx(hClient);  
  
if (nResult == NNTP_ERROR)  
{  
    TCHAR szError[ASTRING];  
    DWORD dwError = NntpGetLastError();  
    NntpGetErrorString(dwError, szError, ASTRING);  
    fprintf(stderr, "Error %08x: %s\n", dwError, szError);  
    return;  
}  
  
// Get each newsgroup, printing the article range and  
// the name of the group  
bResult = NntpGetFirstGroupEx(hClient, &newsGroup);
```

```
while (bResult)
{
    printf("%I64u %I64u %s\n", newsGroup.nFirstArticle,
        newsGroup.nLastArticle,
        newsGroup.szName);

    bResult = NntpGetNextGroupEx(hClient, &newsGroup);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetFirstArticleEx](#), [NntpGetNextArticleEx](#), [NntpGetNextGroupEx](#), [NntpListArticlesEx](#),
[NntpListGroups](#), [NntpListNewGroups](#), [NntpSelectGroup](#), [NEWSARTICLEEX](#), [NEWSGROUPEX](#)

NntpGetGroupName Function

```
INT WINAPI NntpGetGroupName(  
    HCLIENT hClient,  
    LPTSTR lpszGroupName,  
    INT nMaxLength  
);
```

The **NntpGetGroupName** function returns the name of the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

lpszGroupName

Pointer to a string buffer that will contain the name of the currently selected newsgroup.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the newsgroup name. If no newsgroup has been selected, the function will return a value of zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetGroupTitle](#), [NntpListGroup](#), [NntpListNewGroups](#), [NntpSelectGroup](#)

NntpGetGroupTitle Function

```
INT WINAPI NntpGetGroupTitle(  
    HCLIENT hClient,  
    LPTSTR lpszGroupTitle,  
    INT nMaxLength  
);
```

The **NntpGetGroupTitle** function returns a description of the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

lpszGroupTitle

Pointer to a string buffer that will contain a description of the currently selected newsgroup.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the description. If no newsgroup has been selected, the function will return a value of zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The news server must support the XGTITLE command so that the group description can be obtained when the newsgroup is selected. If this command is not recognized, then no description will be returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csnewsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetGroupName](#), [NntpListGroup](#), [NntpListNewGroups](#), [NntpSelectGroup](#)

NntpGetLastError Function

```
DWORD WINAPI NntpGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **NntpSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **NntpGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **NntpSetLastError(0)** when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or NNTP_ERROR. Those functions which call **NntpSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[NntpGetErrorString](#), [NntpSetLastError](#)

NntpGetMessageIdArticle Function

```
LONG WINAPI NntpGetMessageIdArticle(  
    HCLIENT hClient,  
    LPCTSTR lpszMessageId  
);
```

The **NntpGetMessageIdArticle** function returns the article number associated with a message identifier in the current newsgroup.

Parameters

hClient

Handle to the client session.

lpszMessageId

A pointer to a message identifier string.

Return Value

If the function succeeds, the return value is the article number. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

If the server uses 64-bit article IDs, your application should use the **NntpGetMessageIdArticleEx** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnews10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetMessageIdArticleEx](#), [NntpGetArticleMessageId](#), [NntpListArticles](#)

NntpGetMessageIdArticleEx Function

```
BOOL WINAPI NntpGetMessageIdArticleEx(  
    HCLIENT hClient,  
    LPCTSTR LpszMessageId  
    ULONGLONG * LpnArticleId  
);
```

The **NntpGetMessageIdArticleEx** function returns the article number associated with a message identifier in the current newsgroup.

Parameters

hClient

Handle to the client session.

lpszMessageId

A pointer to a message identifier string.

lpnArticleId

A pointer to an unsigned 64-bit integer which will contain the article ID. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value non-zero. If the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetMessageIdArticleEx** function returns a BOOL (a signed 32-bit integer) to indicate success or failure, and the article count is returned in a variable that is passed by reference to the function. If you are updating your code to use this function, make sure you also change how you check the return value.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetArticleMessageIdEx](#), [NntpListArticlesEx](#)

NntpGetMultiLine Function

```
INT WINAPI NntpGetMultiLine(  
    HCLIENT hClient,  
    LPBOOL lpbMultiLine  
);
```

The **NntpGetMultiLine** function returns the value of the client multi-line flag in the specified boolean parameter.

Parameters

hClient

Handle to the client session.

lpbMultiLine

A pointer to a boolean variable. This variable will be set to the current value of the client's internal multi-line flag.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The multi-line flag is used by the library to determine if multiple lines of data will be returned by the server as the result of a command. Unlike a single line response, which consists of a result code and result string, a multi-line response consists of one or more lines of text, terminated by a special end-of-data marker.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[NntpCommand](#), [NntpGetResultCode](#), [NntpGetResultString](#), [NntpSetMultiLine](#)

NntpGetNextArticle Function

```
BOOL WINAPI NntpGetNextArticle(  
    HCLIENT hClient,  
    LPNEWSARTICLE lpArticle  
);
```

The **NntpGetNextArticle** function returns information about the next article in the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

lpArticle

A pointer to a [NEWSARTICLE](#) structure which will contain information about the next available article in the currently selected directory.

Return Value

If the function succeeds, the return value is non-zero. If there are no more articles in the current newsgroup, or the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetNextArticle** function returns information about the next available article in the currently selected newsgroup. This function is used in conjunction with the **NntpGetFirstArticle** function to enumerate all of the articles in the newsgroup. Typically this is used to provide the user with a list of articles to access.

While the articles in the newsgroup are being listed, the client cannot retrieve the contents of a specific article. For example, the **NntpGetArticle** function cannot be called while inside a loop calling **NntpGetNextArticle**. The client should store those articles which it wants to retrieve in an array, and then once all of the articles have been listed, it can begin calling **NntpGetArticle** for each article number to retrieve the article text.

If the server uses 64-bit article IDs, your application should use the **NntpGetFirstArticleEx** and **NntpGetNextArticleEx** functions.

Example

```
NEWSARTICLE newsArticle;  
BOOL bResult;  
INT nResult;  
  
// List all articles in the current group  
nResult = NntpListArticles(hClient, -1L, -1L);  
  
if (nResult == NNTP_ERROR)  
{  
    TCHAR szError[ASTRING];  
    DWORD dwError = NntpGetLastError();  
  
    NntpGetErrorString(dwError, szError, ASTRING);  
    _ftprintf(stderr, _T("Error %08x: %s\n"), dwError, szError);  
    return;  
}
```

```
// Get each article in the current newsgroup, printing the article
// number and the subject of the article
bResult = NntpGetNextArticle(hClient, &newsArticle);

while (bResult)
{
    _tprintf(_T("%ld %s\n"), newsArticle.nArticleId, newsArticle.szSubject);
    bResult = NntpGetNextArticle(hClient, &newsArticle);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetArticle](#), [NntpGetFirstArticle](#), [NntpGetFirstArticleEx](#), [NntpGetFirstGroup](#),
[NntpGetNextArticleEx](#), [NntpGetNextGroup](#), [NntpListArticles](#), [NntpListGroup](#), [NntpListNewGroups](#),
[NntpSelectGroup](#), [NEWSARTICLE](#), [NEWSGROUP](#)

NntpGetNextArticleEx Function

```
BOOL WINAPI NntpGetNextArticleEx(  
    HCLIENT hClient,  
    LPNEWSARTICLEEX lpArticleEx  
);
```

The **NntpGetNextArticleEx** function returns information about the next article in the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

lpArticle

A pointer to a [NEWSARTICLEEX](#) structure which will contain information about the next available article in the currently selected directory.

Return Value

If the function succeeds, the return value is non-zero. If there are no more articles in the current newsgroup, or the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetNextArticleEx** function returns information about the next available article in the currently selected newsgroup. This function is used in conjunction with the **NntpGetFirstArticleEx** function to enumerate all of the articles in the newsgroup. Typically this is used to provide the user with a list of articles to access. This function provides support for servers which use 64-bit article IDs.

While the articles in the newsgroup are being listed, the client cannot retrieve the contents of a specific article. For example, the **NntpGetArticleEx** function cannot be called while inside a loop calling **NntpGetNextArticleEx**. The client should store those articles which it wants to retrieve in an array, and then once all of the articles have been listed, it can begin calling **NntpGetArticleEx** for each article number to retrieve the article text.

Example

```
NEWSARTICLEEX newsArticle;  
BOOL bResult;  
INT nResult;  
  
// List all articles in the current group  
nResult = NntpListArticlesEx(hClient, (ULONGLONG)-1, (ULONGLONG)-1);  
  
if (nResult == NNTP_ERROR)  
{  
    TCHAR szError[ASTRING];  
    DWORD dwError = NntpGetLastError();  
  
    NntpGetErrorString(dwError, szError, ASTRING);  
    _ftprintf(stderr, _T("Error %08x: %s\n"), dwError, szError);  
    return;  
}
```

```
// Get each article in the current newsgroup, printing the article
// number and the subject of the article
bResult = NntpGetNextArticleEx(hClient, &newsArticle);

while (bResult)
{
    _tprintf(_T("%I64u %s\n"), newsArticle.nArticleId, newsArticle.szSubject);
    bResult = NntpGetNextArticleEx(hClient, &newsArticle);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetArticleEx](#), [NntpGetFirstArticleEx](#), [NntpGetFirstGroupEx](#), [NntpGetNextGroupEx](#),
[NntpListArticlesEx](#), [NntpListGroups](#), [NntpListNewGroups](#), [NntpSelectGroup](#), [NEWSARTICLEEX](#),
[NEWSGROUPEX](#)

NntpGetNextGroup Function

```
BOOL WINAPI NntpGetNextGroup(  
    HCLIENT hClient,  
    LPNEWSGROUP lpGroup  
);
```

The **NntpGetNextGroup** function returns information about the next available newsgroup.

Parameters

hClient

Handle to the client session.

lpGroup

A pointer to a [NEWSGROUP](#) structure which will contain information about the next available newsgroup.

Return Value

If the function succeeds, the return value is non-zero. If there are no more newsgroups available, or the function fails, the return value is zero. To get extended error information, call

NntpGetLastError.

Remarks

The **NntpGetNextGroup** function returns information about the next newsgroup on the server. This function is used in conjunction with the **NntpGetFirstGroup** function to enumerate all of the available newsgroups. Typically this is used to provide the user with a list of newsgroups to select.

While the the newsgroups are being listed, the client cannot select a newsgroup or retrieve the contents of a specific article. The client should store those newsgroups which it wants to retrieve articles from, and then once all of the newsgroups have been listed, it can then select each newsgroup and retrieve the available articles from that group.

If the server uses 64-bit article IDs, your application should use the **NntpGetFirstGroupEx** and **NntpGetNextGroupEx** functions.

Example

```
NEWSGROUP newsGroup;  
BOOL bResult;  
INT nResult;  
  
// List all available newsgroups  
nResult = NntpListGroup(hClient);  
  
if (nResult == NNTP_ERROR)  
{  
    TCHAR szError[ASTRING];  
    DWORD dwError = NntpGetLastError();  
  
    NntpGetErrorString(dwError, szError, ASTRING);  
    _ftprintf(stderr, _T("Error %08x: %s\n"), dwError, szError);  
    return;  
}  
  
// Get each newsgroup, printing the article range and  
// the name of the group
```

```
bResult = NntpGetFirstGroup(hClient, &newsGroup);

while (bResult)
{
    _tprintf(_T("%ld %ld %s\n"), newsGroup.nFirstArticle,
            newsGroup.nLastArticle,
            newsGroup.szName);

    bResult = NntpGetNextGroup(hClient, &newsGroup);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetFirstArticle](#), [NntpGetFirstGroup](#), [NntpGetFirstGroupEx](#), [NntpGetNextArticle](#),
[NntpGetNextGroupEx](#), [NntpListArticles](#), [NntpListGroups](#), [NntpListNewGroups](#), [NntpSelectGroup](#),
[NEWSARTICLE](#), [NEWSGROUP](#)

NntpGetNextGroupEx Function

```
BOOL WINAPI NntpGetNextGroupEx(  
    HCLIENT hClient,  
    LPNEWSGROUPEX lpGroupEx  
);
```

The **NntpGetNextGroupEx** function returns information about the next available newsgroup.

Parameters

hClient

Handle to the client session.

lpGroup

A pointer to a [NEWSGROUPEX](#) structure which will contain information about the next available newsgroup.

Return Value

If the function succeeds, the return value is non-zero. If there are no more newsgroups available, or the function fails, the return value is zero. To get extended error information, call

NntpGetLastError.

Remarks

The **NntpGetNextGroupEx** function returns information about the next newsgroup on the server. This function is used in conjunction with the **NntpGetFirstGroupEx** function to enumerate all of the available newsgroups. Typically this is used to provide the user with a list of newsgroups to select. This function provides support for servers which use 64-bit article IDs.

While the the newsgroups are being listed, the client cannot select a newsgroup or retrieve the contents of a specific article. The client should store those newsgroups which it wants to retrieve articles from, and then once all of the newsgroups have been listed, it can then select each newsgroup and retrieve the available articles from that group.

Example

```
NEWSGROUPEX newsGroup;  
BOOL bResult;  
INT nResult;  
  
// List all available newsgroups  
nResult = NntpListGroupEx(hClient);  
  
if (nResult == NNTP_ERROR)  
{  
    TCHAR szError[ASTRING];  
    DWORD dwError = NntpGetLastError();  
  
    NntpGetErrorString(dwError, szError, ASTRING);  
    _ftprintf(stderr, _T("Error %08x: %s\n"), dwError, szError);  
    return;  
}  
  
// Get each newsgroup, printing the article range and  
// the name of the group  
bResult = NntpGetFirstGroupEx(hClient, &newsGroup);
```

```
while (bResult)
{
    _tprintf(_T("%I64u %I64u %s\n"), newsGroup.nFirstArticle,
            newsGroup.nLastArticle,
            newsGroup.szName);

    bResult = NntpGetNextGroup(hClient, &newsGroup);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetFirstArticle](#), [NntpGetFirstGroup](#), [NntpGetNextArticle](#), [NntpListArticles](#), [NntpListGroups](#), [NntpListNewGroups](#), [NntpSelectGroup](#), [NEWSARTICLE](#), [NEWSGROUP](#)

NntpGetResultCode Function

```
INT WINAPI NntpGetResultCode(  
    HCLIENT hClient  
);
```

The **NntpGetResultCode** function reads the result code returned by the server in response to a command. The result code is a three-digit numeric code, and indicates if the operation succeeded, failed or requires additional action by the client.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

Result codes are three-digit numeric values returned by the server. They may be broken down into the following ranges:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpCommand](#), [NntpGetMultiLine](#), [NntpGetResultString](#), [NntpSetMultiLine](#)

NntpGetResultString Function

```
INT WINAPI NntpGetResultString(  
    HCLIENT hClient,  
    LPTSTR lpszResult,  
    INT nMaxLength  
);
```

The **NntpGetResultString** function returns the last message sent by the server along with the result code.

Parameters

hClient

Handle to the client session.

lpszResult

A pointer to the buffer that will contain the result string returned by the server.

nMaxLength

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetResultString** function is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **NntpGetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpCommand](#), [NntpGetMultiLine](#), [NntpGetResultCode](#), [NntpSetMultiLine](#)

NntpGetSecurityInformation Function

```
BOOL WINAPI NntpGetSecurityInformation(  
    HCLIENT hClient,  
    LPSECURITYINFO lpSecurityInfo  
);
```

The **NntpGetSecurityInformation** function returns security protocol, encryption and certificate information about the current client connection.

Parameters

hClient

Handle to the client session.

lpSecurityInfo

A pointer to a [SECURITYINFO](#) structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**.

Remarks

This function is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value `SECURITY_PROTOCOL_NONE`.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;  
  
securityInfo.dwSize = sizeof(SECURITYINFO);  
if (NntpGetSecurityInformation(hClient, &securityInfo))  
{  
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)  
    {  
        MessageBox(NULL, _T("The connection is not secure"),  
                    _T("Connection"), MB_OK);  
    }  
    else  
    {  
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)  
        {  
            MessageBox(NULL, _T("The connection is secure"),  
                        _T("Connection"), MB_OK);  
        }  
        else  
        {  
            MessageBox(NULL, _T("The server certificate not valid"),  
                        _T("Connection"), MB_OK);  
        }  
    }  
}
```

```
}  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpConnect](#), [NntpDisconnect](#), [SECURITYINFO](#)

NntpGetStatus Function

```
INT WINAPI NntpGetStatus(  
    HCLIENT hClient  
);
```

The **NntpGetStatus** function returns the current status of the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetStatus** function returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Constant	Description
1	NNTP_STATUS_IDLE	The client is current idle and not sending or receiving data.
2	NNTP_STATUS_CONNECT	The client is establishing a connection with the server.
3	NNTP_STATUS_READ	The client is reading data from the server.
4	NNTP_STATUS_WRITE	The client is writing data to the server.
5	NNTP_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csntv10.lib

See Also

[NntpIsBlocking](#), [NntpIsConnected](#), [NntpIsReadable](#), [NntpIsWritable](#)

NntpGetTimeout Function

```
INT WINAPI NntpGetTimeout(  
    HCLIENT hClient  
);
```

The **NntpGetTimeout** function returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

See Also

[NntpSetTimeout](#)

NntpGetTransferStatus Function

```
INT WINAPI NntpGetTransferStatus(  
    HCLIENT hClient,  
    LPNNTPTRANSFERSTATUS lpStatus  
);
```

The **NntpGetTransferStatus** function returns information about the current news article transfer in progress.

Parameters

hClient

Handle to the client session.

lpStatus

A pointer to an [NNTPTRANSFERSTATUS](#) structure which contains information about the status of the current article transfer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `NNTP_ERROR`. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetTransferStatus** function returns information about the current data transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If no article is currently being retrieved or submitted to the server, this function will return the status of the last successful data transfer made by the client.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

If the server uses 64-bit article IDs, your application should use the **NntpGetTransferStatusEx** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csnwsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpEnableEvents](#), [NntpGetTransferStatusEx](#), [NntpRegisterEvent](#), [NNTPTRANSFERSTATUS](#)

NntpGetTransferStatusEx Function

```
INT WINAPI NntpGetTransferStatusEx(  
    HCLIENT hClient,  
    LPNNTPTTRANSFERSTATUSEX lpStatusEx  
);
```

The **NntpGetTransferStatusEx** function returns information about the current news article transfer in progress.

Parameters

hClient

Handle to the client session.

lpStatus

A pointer to an [NNTPTTRANSFERSTATUSEX](#) structure which contains information about the status of the current article transfer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `NNTP_ERROR`. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpGetTransferStatusEx** function returns information about the current data transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. This function provides support for servers which use 64-bit article IDs. If no article is currently being retrieved or submitted to the server, this function will return the status of the last successful data transfer made by the client.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csnwsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpEnableEvents](#), [NntpRegisterEvent](#), [NNTPTTRANSFERSTATUSEX](#)

NntpInitialize Function

```
BOOL WINAPI NntpInitialize(  
    LPCTSTR lpszLicenseKey,  
    LPINITDATA lpData  
);
```

The **NntpInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpData

Pointer to an INITDATA data structure. This parameter may be NULL if the initialization data for the library is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **NntpGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must insure that this function is called with a valid license key.

If the *lpData* argument is specified, it must point to an INITDATA structure which has been initialized by setting the *dwSize* member to the size of the structure. All other structure members should be set to zero. If the function is successful, then the INITDATA structure will be filled with identifying information about the library.

Although it is only required that **NntpInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **NntpUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **NntpInitialize** or **NntpUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

NntpAsyncConnect, NntpConnect, NntpDisconnect, NntpUninitialize

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

NntpIsBlocking Function

```
BOOL WINAPI NntpIsBlocking(  
    HCLIENT hClient  
);
```

The **NntpIsBlocking** function is used to determine if the client is currently performing a blocking operation.

Parameters

hClient

Handle to the client session.

Return Value

If the client is performing a blocking operation, the function returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the function returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking function may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **NntpIsBlocking** function can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[NntpCancel](#), [NntpCommand](#), [NntpIsConnected](#), [NntpIsReadable](#), [NntpIsWritable](#)

NntpIsConnected Function

```
BOOL WINAPI NntpIsConnected(  
    HCLIENT hClient  
);
```

The **NntpIsConnected** function is used to determine if the client is currently connected to a server.

Parameters

hClient

Handle to the client session.

Return Value

If the client is connected to a server, the function returns a non-zero value. If the client is not connected, or the client handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpIsBlocking](#), [NntpIsReadable](#), [NntpIsWritable](#)

NntpIsReadable Function

```
BOOL WINAPI NntpIsReadable(  
    HCLIENT hClient,  
    INT nTimeout,  
    LPDWORD lpdwAvail  
);
```

The **NntpIsReadable** function is used to determine if data is available to be read from the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the function returns a non-zero value. If the client cannot read any data, the function returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

See Also

[NntpGetStatus](#), [NntpIsBlocking](#), [NntpIsConnected](#), [NntpIsWritable](#), [NntpRead](#)

NntpIsWritable Function

```
BOOL WINAPI NntpIsWritable(  
    HCLIENT hClient,  
    INT nTimeout  
);
```

The **NntpIsWritable** function is used to determine if data can be written to the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the function returns a non-zero value. If the client cannot write any data, the function returns zero.

Remarks

Although this function can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csnews10.lib

See Also

[NntpGetStatus](#), [NntpIsBlocking](#), [NntpIsConnected](#), [NntpIsReadable](#), [NntpWrite](#)

NntpListArticles Function

```
INT WINAPI NntpListArticles(  
    HCLIENT hClient,  
    LONG nFirstArticle,  
    LONG nLastArticle  
);
```

The **NntpListArticles** function returns a list of articles in the currently selected newsgroup, within the specified article range.

Parameters

hClient

Handle to the client session.

nFirstArticle

The first newsgroup article to be returned in the list. If this value is -1, the list will begin with the first available article in the newsgroup.

nLastArticle

The last newsgroup article to be returned in the list. If the value is -1, the list will end with the last available article in the newsgroup.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

It is possible that there will be gaps in the articles within the range of the first and last articles in the newsgroup. This may be due to a message being canceled or expired. Use the **NntpGetFirstArticle** and **NntpGetNextArticle** functions to read the list of articles returned by the server.

If the server uses 64-bit article IDs, your application should use the **NntpListArticlesEx** function.

Example

```
NEWSARTICLE newsArticle;  
BOOL bResult;  
INT nResult;  
  
// List all articles in the current group  
nResult = NntpListArticles(hClient, -1L, -1L);  
  
if (nResult == NNTP_ERROR)  
{  
    TCHAR szError[ASTRING];  
    DWORD dwError = NntpGetLastError();  
  
    NntpGetErrorString(dwError, szError, ASTRING);  
    _ftprintf(stderr, _T("Error %08x: %s\n"), dwError, szError);  
    return;  
}  
  
// Get each article in the current newsgroup, printing the article  
// number and the subject of the article
```

```
bResult = NntpGetNextArticle(hClient, &newsArticle);
while (bResult)
{
    _tprintf(_T("%ld %s\n"), newsArticle.nArticleId, newsArticle.szSubject);
    bResult = NntpGetNextArticle(hClient, &newsArticle);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetArticleRange](#), [NntpGetCurrentArticle](#), [NntpGetFirstArticle](#), [NntpGetNextArticle](#),
[NntpListGroup](#)s, [NntpListArticlesEx](#), [NntpListNewGroups](#)

NntpListGroup Function

```
INT WINAPI NntpListGroup(  
    HCLIENT hClient  
);
```

The **NntpListGroup** function instructs the server to begin sending a list of newsgroups back to the client.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpListGroup** function is used in conjunction with the **NntpGetFirstGroup** and **NntpGetNextGroup** functions to enumerate all of the available newsgroups. Typically this is used to provide the user with a list of newsgroups to select. To list only those newsgroups which have been added since a certain date, use the **NntpListNewGroups** function.

While the newsgroups are being listed, the client cannot select a newsgroup or retrieve the contents of a specific article. The client should store those newsgroups which it wants to retrieve articles from, and then once all of the newsgroups have been listed, it can then select each newsgroup and retrieve the available articles from that group.

Example

```
NEWSGROUP newsGroup;  
BOOL bResult;  
INT nResult;  
  
// List all available newsgroups  
nResult = NntpListGroup(hClient);  
  
if (nResult == NNTP_ERROR)  
{  
    TCHAR szError[ASTRING];  
    DWORD dwError = NntpGetLastError();  
    NntpGetErrorString(dwError, szError, ASTRING);  
    fprintf(stderr, "Error %08x: %s\n", dwError, szError);  
    return;  
}  
  
// Get each newsgroup, printing the article range and  
// the name of the group  
bResult = NntpGetFirstGroup(hClient, &newsGroup);  
while (bResult)  
{  
    printf("%ld %ld %s\n", newsGroup.nFirstArticle,  
        newsGroup.nLastArticle,  
        newsGroup.szName);  
    bResult = NntpGetNextGroup(hClient, &newsGroup);  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetArticleRange](#), [NntpGetFirstGroup](#), [NntpGetNextGroup](#), [NntpListNewGroups](#),
[NntpSelectGroup](#)

NntpListNewGroups Function

```
INT WINAPI NntpListNewGroups(  
    HCLIENT hClient,  
    LPCTSTR lpszLastUpdated,  
    BOOL bLocalTime  
);
```

The **NntpListNewGroups** function instructs the server to begin sending a list of newsgroups that were created since the specified date.

Parameters

hClient

Handle to the client session.

lpszLastUpdated

Pointer to a string which specifies the date and time that the list of newsgroups were last retrieved from the server. This parameter may be NULL or an empty string, in which case all available newsgroups will be listed by the server.

bLocalTime

A boolean value which indicates if the time specified in the *lpszLastUpdated* parameter is for the current timezone. If the value is non-zero, the time is assumed to be in the local timezone. If the value is zero, the time is assumed to be in Coordinated Universal Time (UTC).

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpListNewGroups** function is used in conjunction with the **NntpGetFirstGroup** and **NntpGetNextGroup** functions to enumerate all of the newsgroups that were added to the server since a specific date and time. Typically this is used to provide the user with a list of updated newsgroups to select. To list all of the newsgroups available on the server, use the **NntpListGroup** function.

While the newsgroups are being listed, the client cannot select a newsgroup or retrieve the contents of a specific article. The client should store those newsgroups which it wants to retrieve articles from, and then once all of the newsgroups have been listed, it can then select each newsgroup and retrieve the available articles from that group.

Example

```
NEWSGROUP newsGroup;  
LPCTSTR lpszUpdated = _T("1/1/2004 12:00 AM");  
BOOL bResult;  
INT nResult;  
  
// List all newsgroups that were added after a  
// specific date  
nResult = NntpListNewGroups(hClient, lpszUpdated, TRUE);  
  
if (nResult == NNTP_ERROR)  
{  
    TCHAR szError[ASTRING];
```

```
    DWORD dwError = NntpGetLastError();
    NntpGetErrorString(dwError, szError, ASTRING);
    fprintf(stderr, "Error %08x: %s\n", dwError, szError);
    return;
}

// Get each newsgroup, printing the article range and
// the name of the group
bResult = NntpGetFirstGroup(hClient, &newsGroup);
while (bResult)
{
    printf("%ld %ld %s\n", newsGroup.nFirstArticle,
           newsGroup.nLastArticle,
           newsGroup.szName);
    bResult = NntpGetNextGroup(hClient, &newsGroup);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetArticleRange](#), [NntpGetFirstGroup](#), [NntpGetNextGroup](#), [NntpListGroup](#), [NntpSelectGroup](#)

NntpOpenArticle Function

```
INT WINAPI NntpOpenArticle(  
    HCLIENT hClient,  
    LONG nArticleId  
);
```

The **NntpOpenArticle** function opens the specified article in the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

nArticleId

An integer value that specifies which article in the current newsgroup to retrieve. This value may be zero, which specifies that the current article should be opened.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpOpenArticle** function sends a request to begin returning the contents of the specified article. The **NntpRead** function is used to read the article contents after it has been opened. When the complete article has been read, the **NntpCloseArticle** function must be called to close the article and complete the request.

It is recommended most applications use the **NntpGetArticle** function, which will retrieve the complete article in a single function call. This function is typically only used if the application needs to modify the contents of the article as it is being read. If you wish to download an article and store it in a file on the local system, use the **NntpStoreArticle** function.

If the server uses 64-bit article IDs, your application should use the **NntpOpenArticleEx** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[NntpCloseArticle](#), [NntpGetArticle](#), [NntpGetArticleRange](#), [NntpGetCurrentArticle](#), [NntpListArticles](#), [NntpOpenArticleByMessageId](#), [NntpOpenNextArticle](#), [NntpOpenPreviousArticle](#), [NntpRead](#), [NntpStoreArticle](#)

NntpOpenArticleEx Function

```
INT WINAPI NntpOpenArticleEx(  
    HCLIENT hClient,  
    ULONGLONG nArticleId  
);
```

The **NntpOpenArticle** function opens the specified article in the currently selected newsgroup.

Parameters

hClient

Handle to the client session.

nArticleId

An unsigned 64-bit integer that specifies which article in the current newsgroup to retrieve. This value may be zero, which specifies that the current article should be opened.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpOpenArticleEx** function sends a request to begin returning the contents of the specified article. This function provides support for servers which use 64-bit article IDs. The **NntpRead** function is used to read the article contents after it has been opened. When the complete article has been read, the **NntpCloseArticle** function must be called to close the article and complete the request.

It is recommended most applications use the **NntpGetArticleEx** function, which will retrieve the complete article in a single function call. This function is typically only used if the application needs to modify the contents of the article as it is being read. If you wish to download an article and store it in a file on the local system, use the **NntpStoreArticleEx** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csntsv10.lib

See Also

[NntpCloseArticle](#), [NntpGetArticle](#), [NntpGetArticleRangeEx](#), [NntpGetCurrentArticleEx](#), [NntpListArticlesEx](#), [NntpOpenArticleByMessageId](#), [NntpOpenNextArticle](#), [NntpOpenPreviousArticle](#), [NntpRead](#), [NntpStoreArticleEx](#)

NntpOpenArticleByMessageId Function

```
INT WINAPI NntpOpenArticleByMessageId(  
    HCLIENT hClient,  
    LPCTSTR lpszMessageId  
);
```

The **NntpOpenArticleByMessageId** function opens the article specified by the message identifier string.

Parameters

hClient

Handle to the client session.

lpszMessageId

Pointer to a string which contains the message identifier for the article in the current newsgroup.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csnews10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpCloseArticle](#), [NntpGetArticle](#), [NntpGetArticleRange](#), [NntpGetCurrentArticle](#), [NntpListArticles](#), [NntpOpenArticle](#), [NntpOpenNextArticle](#), [NntpOpenPreviousArticle](#), [NntpRead](#)

NntpOpenNextArticle Function

```
INT WINAPI NntpOpenNextArticle(  
    HCLIENT hClient  
);
```

The **NntpOpenNextArticle** function opens the next available article in the current newsgroup.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[NntpCloseArticle](#), [NntpGetArticle](#), [NntpGetArticleRange](#), [NntpGetCurrentArticle](#), [NntpListArticles](#), [NntpOpenArticleByMessageId](#), [NntpOpenPreviousArticle](#), [NntpRead](#)

NntpOpenPreviousArticle Function

```
INT WINAPI NntpOpenPreviousArticle(  
    HCLIENT hClient  
);
```

The **NntpOpenPreviousArticle** function opens the previous article in the current newsgroup.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[NntpCloseArticle](#), [NntpGetArticle](#), [NntpGetArticleRange](#), [NntpGetCurrentArticle](#), [NntpListArticles](#), [NntpOpenArticleByMessageId](#), [NntpOpenNextArticle](#), [NntpRead](#)

NntpPostArticle Function

```
INT WINAPI NntpPostArticle(  
    HCLIENT hClient,  
    LPCTSTR lpBuffer,  
    DWORD dwLength,  
    DWORD dwReserved  
);
```

The **NntpPostArticle** function post the contents of the specified buffer to the server as a new article in the current newsgroup.

Parameters

hClient

Handle to the client session.

lpBuffer

A pointer to a character buffer which contains the article to be posted to the currently selected newsgroup.

dwLength

Specifies the length of the string which contains the article. If this parameter is -1, the actual length of the string is calculated by searching the buffer for a terminating null byte.

dwReserved

Reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpPostArticle** function is used to post the contents of the specified buffer to the server as a new article in the current newsgroup. Not all newsgroups permit new articles to be posted, and some newsgroups may require that you email the article to a moderator for approval instead of posting directly to the group. It may be required that the client authenticate itself using the **NntpAuthenticate** function prior to posting the article.

A news article is similar to an email message in that it contains one or more header fields, followed by an empty line, followed by the body of the article. Each line of text should be terminated by a carriage return/linefeed sequence of characters. The Mail Message library can be used to compose a message if needed. Note that the article header must contain a header field named "Newsgroups" with a value that specifies the newsgroup or newsgroups the article is being posted to. If this header field is missing, the news server will reject the article.

This function will cause the current thread to block until the transfer has completed, a timeout occurs or the transfer is canceled. During the transfer, the HTTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **NntpEnableEvents**, or by registering a callback function using the **NntpRegisterEvent** function.

To determine the current status of a transfer while it is in progress, use the **NntpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[NntpGetTransferStatus](#), [NntpIsBlocking](#), [NntpIsWritable](#), [NntpRead](#), [NntpReset](#), [NntpWrite](#)

NntpRead Function

```
INT WINAPI NntpRead(  
    HCLIENT hClient,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **NntpRead** function reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hClient

Handle to the client session.

lpBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

When **NntpRead** is called and the client is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

See Also

[NntpCloseArticle](#), [NntpIsBlocking](#), [NntpIsConnected](#), [NntpIsReadable](#), [NntpOpenArticle](#), [NntpWrite](#)

NntpRegisterEvent Function

```
INT WINAPI NntpRegisterEvent(  
    HCLIENT hClient,  
    UINT nEvent,  
    NNTPEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

The **NntpRegisterEvent** function registers an event handler for the specified event.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Constant	Description
NNTP_EVENT_CONNECT	The connection to the server has completed.
NNTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
NNTP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
NNTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
NNTP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
NNTP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
NNTP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
NNTP_EVENT_LISTARTICLE	The client is retrieving a list of articles from the server and information about another article has been

	processed. The client should use this event to retrieve the next available article from the server.
NNTP_EVENT_LASTARTICLE	The client is retrieving a list of articles from the server and all of the available articles have been processed. This event is generated when there are no further articles.
NNTP_EVENT_LISTGROUP	The client is retrieving a list of newsgroups from the server and information about another group has been processed. The client should use this event to retrieve the next available newsgroup from the server.
NNTP_EVENT_LASTGROUP	The client is retrieving a list of newsgroups from the server and all of the available groups have been processed. This event is generated when there are no further newsgroups.
NNTP_EVENT_PROGRESS	The client is in the process of sending or receiving data from the server. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **NntpEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpRegisterEvent** function associates a callback function with a specific event. The event handler is an **NntpEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This function is typically used to register an event handler that is invoked while a news article is being uploaded or downloaded. The NNTP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one NNTP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

See Also

[NntpDisableEvents](#), [NntpEnableEvents](#), [NntpEventProc](#), [NntpFreezeEvents](#)

NntpReset Function

```
INT WINAPI NntpReset(  
    HCLIENT hClient  
);
```

The **NntpReset** function resets the client state and resynchronizes with the server. This function is typically called after an unexpected error has occurred, or an operation has been canceled.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The client cannot be reset while it is in a blocked state.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

See Also

[NntpCancel](#), [NntpIsBlocking](#)

NntpSelectGroup Function

```
INT WINAPI NntpSelectGroup(  
    HCLIENT hClient,  
    LPCTSTR lpszGroupName  
);
```

The **NntpSelectGroup** function selects the specified newsgroup from which articles will be retrieved.

Parameters

hClient

Handle to the client session.

lpszGroupName

Pointer to a string which specifies the newsgroup to be selected. This value may be NULL, in which case the current newsgroup is unchanged, but the article count is updated.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

This function selects the newsgroup and obtains a description and the first and last article numbers for that group.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetGroupName](#), [NntpGetGroupTitle](#), [NntpListGroup](#), [NntpListNewGroups](#)

NntpSetLastError Function

```
VOID WINAPI NntpSetLastError(  
    DWORD dwErrorCode  
);
```

The **NntpSetLastError** function sets the last error code for the current thread. This function is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or NNTP_ERROR. Those functions which call **NntpSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **NntpGetLastError** function. The use of **NntpGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csntools10.lib

See Also

[NntpGetErrorString](#), [NntpGetLastError](#)

NntpSetMultiLine Function

```
INT WINAPI NntpSetMultiLine(  
    HCLIENT hClient,  
    BOOL bMultiLine  
);
```

The **NntpSetMultiLine** function sets the client multi-line flag into the specified value.

Parameters

hClient

Handle to the client session.

bMultiLine

A boolean flag which determines if the client is processing multiple lines of data as the result of a command.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The multi-line flag is used by the library to determine if multiple lines of data will be returned by the server as the result of a command. Unlike a single line response, which consists of a result code and result string, a multi-line response consists of one or more lines of text, terminated by a special end-of-data marker.

The **NntpSetMultiLine** function should only be used in conjunction with the **NntpCommand** function. If a command is issued which would result in multiple lines of output, the multi-line flag must be set TRUE. The multi-line flag must be set after each command, since it is reset to FALSE with each command that is sent to the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpCommand](#), [NntpGetMultiLine](#), [NntpGetResultCode](#), [NntpGetResultString](#)

NntpSetTimeout Function

```
INT WINAPI NntpSetTimeout(  
    HCLIENT hClient,  
    INT nTimeout  
);
```

The **NntpSetTimeout** function sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csntools10.lib

See Also

[NntpConnect](#), [NntpGetTimeout](#)

NntpStoreArticle Function

```
INT WINAPI NntpStoreArticle(  
    HCLIENT hClient,  
    LONG nArticleId,  
    LPCTSTR lpszFileName  
);
```

The **NntpStoreMessage** function retrieves an article from the current newsgroup and stores it in a local file.

Parameters

hClient

Handle to the client session.

nArticleId

An integer value that specifies the article to be downloaded.

lpszFileName

Pointer to a string which specifies the file that the article will be stored in. If the file does not exist, it will be created. If the file does exist, it will be overwritten with the contents of the article.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpStoreArticle** function provides a method of retrieving and storing an article on the local system. The contents of the article is stored as a text file, using the specified file name. This function always causes the caller to block until the entire message has been retrieved, even if the client has been put in asynchronous mode. If you wish to download the article to memory rather than a file, use the **NntpGetArticle** function.

If event handling is enabled, the NNTP_EVENT_PROGRESS event will fire periodically during the transfer of the article to the local system. An application can determine how much of the article has been retrieved by calling the **NntpGetTransferStatus** function.

If the server uses 64-bit article IDs, your application should use the **NntpStoreArticleEx** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetArticle](#), [NntpGetArticleHeaders](#), [NntpGetTransferStatus](#), [NntpStoreArticleEx](#)

NntpStoreArticleEx Function

```
INT WINAPI NntpStoreArticleEx(  
    HCLIENT hClient,  
    ULONGLONG nArticleId,  
    LPCTSTR lpszFileName,  
    DWORD dwReserved  
);
```

The **NntpStoreMessageEx** function retrieves an article from the current newsgroup and stores it in a local file.

Parameters

hClient

Handle to the client session.

nArticleId

An unsigned 64-integer value which specifies the article to be downloaded.

lpszFileName

Pointer to a string which specifies the file that the article will be stored in. If the file does not exist, it will be created. If the file does exist, it will be overwritten with the contents of the article.

dwReserved

Number of the article to retrieve. This value must be greater than zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The **NntpStoreArticleEx** function provides a method of retrieving and storing an article on the local system. This function provides support for servers which use 64-bit article IDs.

The contents of the article is stored as a text file, using the specified file name. This function always causes the caller to block until the entire message has been retrieved, even if the client has been put in asynchronous mode. If you wish to download the article to memory rather than a file, use the **NntpGetArticle** function.

If event handling is enabled, the NNTP_EVENT_PROGRESS event will fire periodically during the transfer of the article to the local system. An application can determine how much of the article has been retrieved by calling the **NntpGetTransferStatusEx** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetArticleEx](#), [NntpGetArticleHeadersEx](#), [NntpGetTransferStatusEx](#)

NntpUninitialize Function

```
VOID WINAPI NntpUninitialize();
```

The **NntpUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **NntpInitialize** call before it can call any of the other library functions. When it has completed the use of library, the application must call **NntpUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There must be a call to **NntpUninitialize** for every successful call to **NntpInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpDisconnect](#), [NntpInitialize](#)

NntpWrite Function

```
INT WINAPI NntpWrite(  
    HCLIENT hClient,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **NntpWrite** function sends the specified number of bytes to the server.

Parameters

hClient

Handle to the client session.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is NNTP_ERROR. To get extended error information, call **NntpGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the NNTP_EVENT_WRITE asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpCreateArticle](#), [NntpIsBlocking](#), [NntpRead](#)

Network News Transfer Protocol Data Structures

- INITDATA
- NEWSARTICLE
- NEWSARTICLEX
- NEWSGROUP
- NEWSGROUPEX
- NNTPTRANSFERSTATUS
- NNTPTRANSFERSTATUSEX
- SECURITYCREDENTIALS
- SECURITYINFO
- SYSTEMTIME

INITDATA Structure

This structure contains information about the SocketTools library when the initialization function returns.

```
typedef struct _INITDATA
{
    DWORD      dwSize;
    DWORD      dwVersionMajor;
    DWORD      dwVersionMinor;
    DWORD      dwVersionBuild;
    DWORD      dwOptions;
    DWORD_PTR  dwReserved1;
    DWORD_PTR  dwReserved2;
    TCHAR      szDescription[128];
} INITDATA, *LPINITDATA;
```

Members

dwSize

Size of this structure. This structure member must be set to the size of the structure prior to calling the initialization function. Failure to do so will cause the function to fail.

dwVersionMajor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the major version number for the library.

dwVersionMinor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the minor version number for the library.

dwVersionBuild

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the build number for the library.

dwOptions

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved1

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved2

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

szDescription

A null terminated string which describes the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

NEWSARTICLE Structure

This structure is used by the **NntpGetFirstArticle** and **NntpGetNextArticle** functions to return information about articles in the currently selected directory. If the server uses 64-bit article IDs your application should use the [NEWSARTICLEEX](#) structure and related functions.

```
typedef struct _NEWSARTICLE
{
    LONG        nArticleId;
    LONG        nBytes;
    LONG        nLines;
    TCHAR       szSubject[NNTP_MAXSUBJLEN];
    TCHAR       szAuthor[NNTP_MAXAUTHLEN];
    TCHAR       szMessageId[NNTP_MAXMSGIDLEN];
    TCHAR       szReferences[NNTP_MAXREFLEN];
    SYSTEMTIME  stPosted;
} NEWSARTICLE, *LPNEWSARTICLE;
```

Members

nArticleId

A long integer which specifies the article number.

nBytes

The length of the news article in bytes.

nLines

The length of the news article specified as the number of lines of text.

szSubject

A pointer to a string which specifies the subject of the article.

szAuthor

A pointer to a string which specifies the email address of the user who posted the article.

szMessageId

A pointer to a string which specifies the message ID for the article.

szReferences

A pointer to a string which specifies references to the article.

stPosted

A SYSTEMTIME structure which specifies when the article was posted in Universal Coordinated Time (UTC).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetFirstArticle](#), [NntpGetNextArticle](#), [NntpListArticles](#), [NEWSARTICLE](#), [NEWSGROUP](#)

NEWSARTICLEEX Structure

This structure is used by the **NntpGetFirstArticleEx** and **NntpGetNextArticleEx** functions to return information about articles in the currently selected directory. This structure is intended for use with servers that return 64-bit article IDs.

```
typedef struct _NEWSARTICLEEX
{
    ULONGLONG    nArticleId;
    ULONG        nBytes;
    ULONG        nLines;
    TCHAR        szSubject[NNTP_MAXSUBJLEN];
    TCHAR        szAuthor[NNTP_MAXAUTHLEN];
    TCHAR        szMessageId[NNTP_MAXMSGIDLEN];
    TCHAR        szReferences[NNTP_MAXREFLEN];
    SYSTEMTIME   stPosted;
} NEWSARTICLEEX, *LPNEWSARTICLEEX;
```

Members

nArticleId

An unsigned 64-bit integer which specifies the article number.

nBytes

The length of the news article in bytes.

nLines

The length of the news article specified as the number of lines of text.

szSubject

A pointer to a string which specifies the subject of the article.

szAuthor

A pointer to a string which specifies the email address of the user who posted the article.

szMessageId

A pointer to a string which specifies the message ID for the article.

szReferences

A pointer to a string which specifies references to the article.

stPosted

A SYSTEMTIME structure which specifies when the article was posted in Universal Coordinated Time (UTC).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetFirstArticleEx](#), [NntpGetNextArticleEx](#), [NntpListArticlesEx](#), [NEWSARTICLEEX](#), [NEWSGROUPEX](#)

NEWSGROUP Structure

This structure is used by the **NntpGetFirstGroup** and **NntpGetNextGroup** functions to return information about available newsgroups. If the server uses 64-bit article IDs your application should use the [NEWSGROUPEX](#) structure and related functions.

```
typedef struct _NEWSGROUP
{
    LONG        nFirstArticle;
    LONG        nLastArticle;
    DWORD       dwAccess;
    TCHAR       szName[NNTP_MAXGRPNAMLEN];
} NEWSGROUP, *LPNEWSGROUP;
```

Members

nFirstArticle

A long integer which specifies the article number of the first available article in the newsgroup.

nLastArticle

A long integer which specifies the article number of the last available article in the newsgroup. Note that posted articles may not be contiguous in the range between the first and last article numbers. Some servers may assign numbers in a different order than the articles were posted, or there may be gaps where articles have been removed.

dwAccess

An unsigned integer which specifies the access mode for the group. It may be one of the following values:

Constant	Description
NNTP_GROUP_READONLY	The group is read-only and cannot be modified. Attempts to post articles to the newsgroup will result in an error.
NNTP_GROUP_READWRITE	Articles can be posted to the newsgroup. Even though a newsgroup is read-write, it may require that the client authenticate before being given permission to post articles to the server.
NNTP_GROUP_MODERATED	The newsgroup is moderated and articles can only be posted by the group moderator. To request that an article be posted to the newsgroup, you must email the message to the moderator.

szName

A pointer to a string which specifies the name of the newsgroup.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csnews10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetFirstGroup](#), [NntpGetNextGroup](#), [NEWSARTICLE](#), [NEWSGROUP](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

NEWSGROUPEX Structure

This structure is used by the **NntpGetFirstGroupEx** and **NntpGetNextGroupEx** functions to return information about available newsgroups. This structure is intended for use with servers that return 64-bit article IDs.

```
typedef struct _NEWSGROUPEX
{
    ULONGLONG    nFirstArticle;
    ULONGLONG    nLastArticle;
    DWORD        dwAccess;
    TCHAR        szName[NNTP_MAXGRPNAMELEN];
} NEWSGROUP, *LPNEWSGROUP;
```

Members

nFirstArticle

An unsigned 64-bit integer which specifies the article number of the first available article in the newsgroup.

nLastArticle

An unsigned 64-bit integer which specifies the article number of the last available article in the newsgroup. Note that posted articles may not be contiguous in the range between the first and last article numbers. Some servers may assign numbers in a different order than the articles were posted, or there may be gaps where articles have been removed.

dwAccess

An unsigned integer which specifies the access mode for the group. It may be one of the following values:

Constant	Description
NNTP_GROUP_READONLY	The group is read-only and cannot be modified. Attempts to post articles to the newsgroup will result in an error.
NNTP_GROUP_READWRITE	Articles can be posted to the newsgroup. Even though a newsgroup is read-write, it may require that the client authenticate before being given permission to post articles to the server.
NNTP_GROUP_MODERATED	The newsgroup is moderated and articles can only be posted by the group moderator. To request that an article be posted to the newsgroup, you must email the message to the moderator.

szName

A pointer to a string which specifies the name of the newsgroup.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnwsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NntpGetFirstGroupEx](#), [NntpGetNextGroupEx](#), [NEWSARTICLEEX](#), [NEWSGROUPEX](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

NNTPTRANSFERSTATUS Structure

This structure is used by the **NntpGetTransferStatus** function to return information about an article transfer in progress. If the server uses 64-bit article IDs your application should use the [NNTPTRANSFERSTATUSEX](#) structure and related functions.

```
typedef struct _NNTPTRANSFERSTATUS
{
    LONG    nArticleId;
    DWORD   dwBytesTotal;
    DWORD   dwBytesCopied;
    DWORD   dwBytesPerSecond;
    DWORD   dwTimeElapsed;
    DWORD   dwTimeEstimated;
} NNTPTRANSFERSTATUS, *LPNNTPTRANSFERSTATUS;
```

Members

nArticleId

A signed integer that specifies the article ID of the current article that is being transferred. If an article is being posted, this value will be zero.

dwBytesTotal

An unsigned integer that specifies the total number of bytes that will be transferred. If the article is being copied from the server to the local host, this is the size of the article on the server. If the article is being posted to the server, it is the size of article on the local system. If the article size cannot be determined, this value will be zero.

dwBytesCopied

An unsigned integer that specifies the total number of bytes that have been copied.

dwBytesPerSecond

An unsigned integer that specifies the average number of bytes that have been copied per second.

dwTimeElapsed

An unsigned integer that specifies the number of seconds that have elapsed since the transfer started.

dwTimeEstimated

An unsigned integer that specifies the estimated number of seconds until the transfer is completed. This is based on the average number of bytes transferred per second.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

See Also

[NntpEnableEvents](#), [NntpGetTransferStatus](#), [NntpRegisterEvent](#), [NNTPTRANSFERSTATUS](#), [NNTPTRANSFERSTATUSEX](#)

NNTPTRANSFERSTATUSEX Structure

This structure is used by the **NntpGetTransferStatusEx** function to return information about an article transfer in progress. This structure is intended for use with servers that return 64-bit article IDs.

```
typedef struct _NNTPTRANSFERSTATUSEX
{
    ULONGLONG nArticleId;
    DWORD     dwBytesTotal;
    DWORD     dwBytesCopied;
    DWORD     dwBytesPerSecond;
    DWORD     dwTimeElapsed;
    DWORD     dwTimeEstimated;
} NNTPTRANSFERSTATUSEX, *LPNNTPTRANSFERSTATUSEX;
```

Members

nArticleId

An unsigned 64-bit integer that specifies the article ID of the current article that is being transferred. If an article is being posted, this value will be zero.

dwBytesTotal

An unsigned integer that specifies the total number of bytes that will be transferred. If the article is being copied from the server to the local host, this is the size of the article on the server. If the article is being posted to the server, it is the size of article on the local system. If the article size cannot be determined, this value will be zero.

dwBytesCopied

An unsigned integer that specifies the total number of bytes that have been copied.

dwBytesPerSecond

An unsigned integer that specifies the average number of bytes that have been copied per second.

dwTimeElapsed

An unsigned integer that specifies the number of seconds that have elapsed since the transfer started.

dwTimeEstimated

An unsigned integer that specifies the estimated number of seconds until the transfer is completed. This is based on the average number of bytes transferred per second.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

See Also

[NntpEnableEvents](#), [NntpGetTransferStatusEx](#), [NntpRegisterEvent](#)

SECURITYCREDENTIALS Structure

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on

	what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that

will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A pointer to a null terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A pointer to a null terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A pointer to a null terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A pointer to a null terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
------------	-------------

CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
----	---

MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
----	--

ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.
------	--

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

A pointer to a null terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used for SSH connections

and should always be NULL when establishing a secure connection using SSL or TLS.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* member should be set to CREDENTIAL_STORE_FILENAME and the *lpzCertStore* member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS#12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the *lpzPassword* member or the library will be unable to access the certificate.

Note that the *lpzUserName* and *lpzPassword* members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

The TLS 1.1 and TLS 1.2 protocols are only supported on Windows 7, Windows Server 2008 R2 and later versions of the platform. If these options are specified and the application is running on Windows XP or Windows Vista, the protocol version will be downgraded to TLS 1.0 for backwards compatibility with those platforms.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

SECURITYINFO Structure

This structure contains information about a secure connection that has been established.

```
typedef struct _SECURITYINFO
{
    DWORD        dwSize;
    DWORD        dwProtocol;
    DWORD        dwCipher;
    DWORD        dwCipherStrength;
    DWORD        dwHash;
    DWORD        dwHashStrength;
    DWORD        dwKeyExchange;
    DWORD        dwCertStatus;
    SYSTEMTIME   stCertIssued;
    SYSTEMTIME   stCertExpires;
    LPCTSTR      lpszCertIssuer;
    LPCTSTR      lpszCertSubject;
    LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to **sizeof(SECURITYINFO)** prior to passing the address of this structure to the function. Note that if this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the protocol that was selected to establish the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The

	<p>correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.</p>
SECURITY_PROTOCOL_TLS10	<p>The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS11	<p>The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS12	<p>The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.</p>

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_CIPHER_RC2	The RC2 block cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
SECURITY_CIPHER_DES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher, using 56-bit

	keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher (also known as the Rijndael cipher) is a fixed block size cipher which use a key size of 128, 192 or 256 bits. This cipher is supported on Windows XP SP3 and later versions of the operating system.
SECURITY_CIPHER_SKIPJACK	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
SECURITY_CIPHER_BLOWFISH	The Blowfish block cipher was selected. This is a variable key length cipher up to 448 bits, using a 64-bit block size.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected. One of the following values will be returned:

Constant	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected. One of the following values will be returned:

--	--

Constant	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography. This key exchange algorithm is only supported on Windows XP SP3 and later versions of the operating system.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server. This member only has meaning for connections using the SSL or TLS protocols. One of the following values will be returned:

Constant	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertIssuer

A pointer to a string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertSubject

A pointer to a string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszFingerprint

A pointer to a string which contains a sequence of hexadecimal values that uniquely identify the server. This member is only used when a connection has been established using the Secure Shell (SSH) protocol.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Unicode: Implemented as Unicode and ANSI versions.

SYSTEMTIME Structure

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include windows.h.

News Feed Library

Retrieve and process the contents of a syndicated news feed.

Reference

- [Functions](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

File Name	CSRSSV10.DLL
Version	10.0.1468.2518
LibID	1F65A283-7BC8-4F5B-B739-D211EAF1CA35
Import Library	CSRSSV10.LIB
Dependencies	None

Overview

Really Simple Syndication (RSS) is a collection of standardized formats that are used to publish information about content that is frequently changed. A news feed is published in XML format, which contains one or more items that includes summary text, hyperlinks to source content and additional metadata that is used to describe the item. News feeds can be used for a variety of purposes, including providing updates for weblogs, news headlines, video and audio content. RSS can also be used for other purposes, such as a software updates, where new updates are listed as items in the feed.

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

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

News Feed Functions

Function	Description
RssCloseFeed	Close the specified news feed and release memory allocated for the channel
RssDisableTrace	Disable logging of network function calls
RssEnableTrace	Enable logging of network function calls to a text file
RssFindItem	Find a specific news feed item using its unique identifier (GUID) property
RssGetErrorString	Return a description for the specified error code
RssGetFirstItem	Return information about the first item in the news feed channel
RssGetItem	Return information about the specified news feed item
RssGetItemCount	Return the number of news feed items in the channel
RssGetItemProperty	Return the value of the specified news feed item property or attribute
RssGetItemText	Return the text description of the specified news feed item
RssGetLastError	Return the last error code
RssGetNextItem	Return information about the next item in the news feed channel
RssInitialize	Initialize the library and validate the specified license key at runtime
RssOpenFeed	Open the specified news feed and return information about the channel
RssParseFeed	Parse the contents of a string and return information about the channel
RssRefreshFeed	Refresh the specified news feed, updating the items in the channel
RssSetLastError	Set the last error code
RssStoreFeed	Store the contents of the specified news feed in an XML formatted text file
RssUninitialize	Terminate use of the library by the application
RssValidateFeed	Validate the contents of the specified news feed, returning the number of items in the feed

RssCloseFeed Function

```
INT WINAPI RssCloseFeed(  
    HCHANNEL hChannel  
);
```

The **RssCloseFeed** function closes the specified news feed and releases memory allocated for the channel.

Parameters

hChannel

Handle to the news feed channel.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `RSS_ERROR`. To get extended error information, call **RssGetLastError**.

Remarks

The **RssCloseFeed** function must be called whenever the application has completed processing the news feed. It is important to note that the memory allocated for the channel will be released when this function is called, which means that any data referenced in the `RSSCHANNEL` and `RSSCHANNELITEM` structures will no longer be valid and must not be used by the application after the feed has been closed.

This function can fail if the feed is currently being updated, such as when the **RssRefreshFeed** function is called. In this case, the channel handle will not be released and the application must attempt to close the feed at a later time.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csrsv10.lib`

See Also

[RssOpenFeed](#) [RssParseFeed](#) [RssStoreFeed](#)

RssDisableTrace Function

```
BOOL WINAPI RssDisableTrace();
```

The **RssDisableTrace** function disables the logging of network function calls.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csrssv10.lib

See Also

[RssEnableTrace](#)

RssEnableTrace Function

```
BOOL WINAPI RssEnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **RssEnableTrace** function enables the logging of network function calls to a text file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace function logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RssDisableTrace](#)

RssFindItem Function

```
INT WINAPI RssFindItem(  
    HCHANNEL hChannel,  
    LPCTSTR lpszValue,  
    DWORD dwOptions,  
    LPRSSCHANNELITEM lpItem  
);
```

The **RssFindItem** function searches for an item in the news feed channel which matches the unique identifier (GUID) value and returns information about that item.

Parameters

hChannel

Handle to the news feed channel.

lpszValue

A pointer to a string which specifies the value of the item being searched for. This value should uniquely identify the item in the feed, and this parameter cannot be an empty string or NULL pointer.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
RSS_FIND_GUID	Search the feed for items with a matching GUID property value. This is the default option, and is the only item property that is guaranteed to be unique in the feed. The search is case-sensitive, requiring that the <i>lpszValue</i> parameter match the property value exactly.
RSS_FIND_LINK	Search the feed for items with a matching link property value. For feeds that do not specify a GUID property, this is the recommended option for searching for an item. The search is not case-sensitive.
RSS_FIND_TITLE	Search the feed for items with a matching title. This option should not be used if you must ensure that the item returned is unique in the feed because there may be multiple items with the same title in the feed. The search is not case-sensitive.
RSS_FIND_PUBDATE	Search the feed for items with a matching publishing date. This option should not be used if you must ensure that the item returned is unique in the feed because more than one item may have the same publishing date. The format of the date string must match the standard format used with the RSS protocol and the match is not case-sensitive.

lpItem

A pointer to a **RSSCHANNELITEM** structure which contains information about the specified item in the news feed. This structure is initialized by the function and the parameter can never be specified as a NULL pointer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `RSS_ERROR`. To get extended error information, call **RssGetLastError**.

Remarks

It is recommended that you use the `RSS_FIND_GUID` option with news feeds that are using version 2.0 or later of the RSS specification. If the feed uses an earlier version, items may not include a GUID property. It is also possible that a feed may omit the GUID property even though it is considered a requirement for the current RSS specification. For the broadest compatibility with all news feeds, an application should not depend on being able to search for a specific news feed item by its GUID.

Only the GUID property is guaranteed to be unique in the feed. If the feed does not specify GUIDs for the news items, the application must use an alternate criteria such as the item hyperlink or publishing date. If there are multiple items that match the *lpzValue* value, the first matching item will be returned.

The data referenced in the **RSSCHANNELITEM** structure should be considered read-only and never modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed, the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csrsvg10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RssGetFirstItem](#), [RssGetItem](#), [RssGetItemProperty](#), [RssGetItemText](#), [RssGetNextItem](#), [RSSCHANNELITEM](#)

RssGetErrorString Function

```
INT WINAPI RssGetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);
```

The **RssGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **RssGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The last-error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csrsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RssGetLastError](#), [RssSetLastError](#)

RssGetFirstItem Function

```
BOOL WINAPI RssGetFirstItem(  
    HCHANNEL hChannel,  
    LPRSSCHANNELITEM lpItem  
);
```

The **RssGetFirstItem** function returns information about the first item in the specified news feed channel.

Parameters

hChannel

Handle to the news feed channel.

lpItem

A pointer to a **RSSCHANNELITEM** structure which contains information about the first item in the news feed. This structure is initialized by the function and the parameter can never be specified as a NULL pointer.

Return Value

If the function succeeds, it returns a non-zero value. If the function fails, the return value is zero. To get extended error information, call **RssGetLastError**.

Remarks

The **RssGetFirstItem** function is used in conjunction with the **RssGetNextItem** function to enumerate the available items in the specified news feed channel. If this function fails, it typically indicates that the channel does not contain any valid news items or that the format of the news feed is invalid.

The data referenced in the **RSSCHANNELITEM** structure should be considered read-only and never modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed, the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RssFindItem](#), [RssGetItem](#), [RssGetItemProperty](#), [RssGetItemText](#), [RssGetNextItem](#), [RSSCHANNELITEM](#)

RssGetItem Function

```
BOOL WINAPI RssGetItem(  
    HCHANNEL hChannel,  
    UINT nItemId,  
    LPRSSCHANNELITEM lpItem  
);
```

The **RssGetItem** function returns information about the specified item in the news feed channel.

Parameters

hChannel

Handle to the news feed channel.

nItemId

An integer value which identifies the news feed item. The first item identifier in the news feed has a value of one, and that value is incremented for each additional item in the feed. If this parameter is zero or specifies a value larger than the number of items in the feed, this function will fail.

lpItem

A pointer to a **RSSCHANNELITEM** structure which contains information about the specified item in the news feed. This structure is initialized by the function and the parameter can never be specified as a NULL pointer.

Return Value

If the function succeeds, it returns a value of zero. If the function fails, the return value is **RSS_ERROR**. To get extended error information, call **RssGetLastError**.

Remarks

The **RssGetItem** function is used to return information about a specific item in the news feed. If this function fails, it typically indicates that the item ID is invalid or that the feed does not contain any valid news items. The **RssGetItemCount** function can be used to determine the number of items contained in the feed channel.

The data referenced in the **RSSCHANNELITEM** structure should be considered read-only and never modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed, the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csrsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RssFindItem](#), [RssGetFirstItem](#), [RssGetItemCount](#), [RssGetItemProperty](#), [RssGetItemText](#), [RssGetNextItem](#), [RSSCHANNELITEM](#)

RssGetItemCount Function

```
INT WINAPI RssGetItemCount(  
    HCHANNEL hChannel  
);
```

The **RssGetItemCount** function returns the number of items in the news feed channel.

Parameters

hChannel

Handle to the news feed channel.

Return Value

If the function succeeds, the return value is the number of items in the news feed. A value of zero indicates that the feed channel is empty. If the function fails, the return value is `RSS_ERROR`. To get extended error information, call **RssGetLastError**.

Remarks

The **RssGetItemCount** function is used to determine the number of items that are contained in the news feed channel, and therefore determine the maximum value of the item identifier which can be used to reference a specific item in the feed. This value is the same as the value specified by the *nItemCount* member of the **RSSCHANNEL** structure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csrsvg10.lib`

See Also

[RssFindItem](#), [RssGetItem](#), [RssGetItemProperty](#), [RssGetItemText](#), [RSSCHANNEL](#)

RssGetItemProperty Function

```
INT WINAPI RssGetItemProperty(  
    HCHANNEL hChannel,  
    UINT nItemId,  
    DWORD dwReserved,  
    LPCTSTR lpszProperty,  
    LPCTSTR lpszAttribute,  
    LPTSTR lpszValue,  
    INT nMaxLength  
);
```

The **RssGetItemProperty** function is used to return the value of a property for the specified item in the news feed channel.

Parameters

hChannel

Handle to the news feed channel.

nItemId

An integer value which identifies the news feed item. The first item identifier in the news feed has a value of one, and that value is incremented for each additional item in the feed. If this parameter is zero or specifies a value larger than the number of items in the feed, this function will fail.

dwReserved

A parameter reserved for future use. This value should always be zero.

lpszProperty

A pointer to a string which specifies the name of the item property. This parameter cannot point to an empty string or specify a NULL pointer.

lpszAttribute

A pointer to a string which specifies the name of an attribute for the property. If this parameter is an empty string or NULL pointer, the function will return the value of the property, rather than an attribute of the specified property.

lpszValue

A pointer to a string buffer which will contain the value of the specified item property or attribute. This string should be large enough to contain the property value. If this parameter is a NULL pointer, it will be ignored and the function will only return the length of value for the specified property.

nMaxLength

The maximum number of characters that may be copied into the property value buffer. If the value of this parameter is zero, then the *lpszValue* parameter is ignored and the function will only return the length of the value for the specified property.

Return Value

If the function succeeds, the return value is the length of the property value string. A return value of zero indicates that the property does not contain any value. If the function fails, the return value is `RSS_ERROR`. To get extended error information, call **RssGetLastError**.

Remarks

The **RssGetItemProperty** function is primarily used with custom item properties that may be used with extensions to the news feed. The standard properties for an item such as the title, link and description can be obtained using **RssGetItem** and related functions. However, if items in the feed contain custom properties that are not part of the standard RSS format, this function can be used to obtain those values.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrssv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RssFindItem](#), [RssGetItem](#), [RssGetItemText](#), [RSSCHANNELITEM](#)

RssGetItemText Function

```
INT WINAPI RssGetItemText(  
    HCHANNEL hChannel,  
    UINT nItemId,  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);
```

The **RssGetItemText** function is used to return a copy of an item's description.

Parameters

hChannel

Handle to the news feed channel.

nItemId

An integer value which identifies the news feed item. The first item identifier in the news feed has a value of one, and that value is incremented for each additional item in the feed. If this parameter is zero or specifies a value larger than the number of items in the feed, this function will fail.

lpszBuffer

A pointer to a string buffer which will contain the value of the item description. If this parameter is a NULL pointer, it will be ignored and the function will only return the length of the item description.

nMaxLength

The maximum number of characters that may be copied into the description buffer. If the value of this parameter is zero, then the *lpszValue* parameter is ignored and the function will only return the length of the item description.

Return Value

If the function succeeds, the return value is the length of the item description. A return value of zero indicates that the item does not have a description. If the function fails, the return value is `RSS_ERROR`. To get extended error information, call **RssGetLastError**.

Remarks

The **RssGetItemText** function is used to obtain a copy of the string that describes the specified item. Typically this is text that provides a summary of the news feed item and is used in conjunction with the item's title and hyperlink to additional content.

The content of an item description is typically either plain text or HTML formatted text. It is the responsibility of the application to display the content in a format is appropriate for the end-user.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csrsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RssFindItem](#), [RssGetItem](#), [RssGetItemProperty](#), [RSSCHANNELITEM](#)

RssGetLastError Function

DWORD WINAPI RssGetLastError();

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **RssSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **RssGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **RssSetLastError(0)** when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CHANNEL or RSS_ERROR. Those functions which call **RssSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrssv10.lib

See Also

[RssGetErrorString](#), [RssSetLastError](#)

RssGetNextItem Function

```
BOOL WINAPI RssGetNextItem(  
    HCHANNEL hChannel,  
    LPRSSCHANNELITEM lpItem  
);
```

The **RssGetNextItem** function returns information about the next item in the specified news feed channel.

Parameters

hChannel

Handle to the news feed channel.

lpItem

A pointer to a **RSSCHANNELITEM** structure which contains information about the next item in the news feed. This structure is initialized by the function and the parameter can never be specified as a NULL pointer.

Return Value

If the function succeeds, it returns a non-zero value. If the function fails, the return value is zero. To get extended error information, call **RssGetLastError**.

Remarks

The **RssGetNextItem** function is used in conjunction with the **RssGetFirstItem** function to enumerate the available items in the specified news feed channel. If this function fails, it typically indicates that there are no more items in the news feed channel.

The data referenced in the **RSSCHANNELITEM** structure should be considered read-only and never modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed, the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrssv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RssFindItem](#), [RssGetFirstItem](#), [RssGetItem](#), [RssGetItemProperty](#), [RssGetItemText](#), [RSSCHANNELITEM](#)

RssInitialize Function

```
BOOL WINAPI RssInitialize(  
    LPCTSTR lpszLicenseKey,  
    LPINITDATA lpData  
);
```

The **RssInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpData

Pointer to an INITDATA data structure. This parameter may be NULL if the initialization data for the library is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **RssGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must insure that this function is called with a valid license key.

If the *lpData* argument is specified, it must point to an INITDATA structure which has been initialized by setting the *dwSize* member to the size of the structure. All other structure members should be set to zero. If the function is successful, then the INITDATA structure will be filled with identifying information about the library.

Although it is only required that **RssInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **RssUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **RssInitialize** or **RssUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RssCloseFeed](#), [RssOpenFeed](#), [RssParseFeed](#), [RssUninitialize](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

RssOpenFeed Function

```
HCHANNEL WINAPI RssOpenFeed(  
    LPCTSTR lpszFeedUrl,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPRSSCHANNEL lpChannel  
);
```

The **RssOpenFeed** function is used to open a news feed and return a handle which can be used to access the individual news items in the feed.

Parameters

lpszFeedUrl

A pointer to a string which specifies the URL for the news feed. To access a news feed on a web server, a standard http or https URL may be specified. To access a file on the local system or network share, a file name or file URL may be specified.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation. This parameter is ignored if the *lpszFeedUrl* parameter specifies a local file name or URL.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
RSS_OPTION_NONE	No additional options are specified and the news feed is processed using relaxed rules when checking the validity of the feed. The library will attempt to automatically compensate for a feed that is malformed or does not strictly conform to the RSS standard.
RSS_OPTION_STRICT	The news feed content should be processed using strict rules to ensure that the feed meets the appropriate RSS standard specification and all feed property values are case-sensitive. By default, relaxed rules are used which allows the application to open a feed that may not strictly conform to the standard specification.

lpChannel

A pointer to an **RSSCHANNEL** structure which contains information about the news feed channel such as the feed title, hyperlink and description. If the parameter is not NULL, the structure is initialized by the function. If the parameter is NULL, it is ignored and no information is returned.

Return Value

If the function succeeds, the return value is a handle to the news feed channel. If the function fails, the return value is INVALID_CHANNEL. To get extended error information, call **RssGetLastError**.

Remarks

A news feed may be local or remote, depending on the URL that is specified. If a local file name or file URL is specified for the feed, then it is opened locally and no network access is required. If an http or https URL is specified, then **RssOpenFeed** will attempt to download the feed from the server and store it temporarily on the local system. Accessing a remote feed requires that the application has permission to establish a connection with the server and will cause the application to block until the feed has been downloaded, the operation times out or an error occurs.

Although the **RssOpenFeed** function will meet the needs of most applications, if you require more complex functionality such as retrieving the feed asynchronously in the background or event notifications for large transfers, you can use the SocketTools [Hypertext Transfer Protocol](#) API to download the news feed and then use the **RssParseFeed** function to parse the contents.

The data referenced in the **RSSCHANNEL** structure should be considered read-only and never modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed, the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrssv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RssCloseFeed](#), [RssInitialize](#), [RssParseFeed](#), [RssRefreshFeed](#), [RssStoreFeed](#), [RssUninitialize](#), [RssValidateFeed](#), [RSSCHANNEL](#)

RssParseFeed Function

```
HCHANNEL WINAPI RssParseFeed(  
    LPCTSTR lpszFeedXml,  
    DWORD dwOptions,  
    LPRSSCHANNEL lpChannel  
);
```

The **RssParseFeed** function is used to parse the contents of a news feed, returning a handle which can be used to access the individual news items in the feed.

Parameters

lpszFeedXml

A pointer to a string which contains the contents of the news feed. The string must contain XML formatted data that conforms to the RSS standard specification. This parameter cannot specify an empty string or a NULL pointer.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
RSS_OPTION_NONE	No additional options are specified.
RSS_OPTION_STRICT	The news feed content should be processed using strict rules to ensure that the feed meets the appropriate RSS standard specification and all feed property values are case-sensitive. By default, relaxed rules are used which allows the application to open a feed that may not strictly conform to the standard specification.

lpChannel

A pointer to an **RSSCHANNEL** structure which contains information about the news feed channel such as the feed title, hyperlink and description. If the parameter is not NULL, the structure is initialized by the function. If the parameter is NULL, it is ignored and no information is returned.

Return Value

If the function succeeds, the return value is a handle to the news feed channel. If the function fails, the return value is **INVALID_CHANNEL**. To get extended error information, call **RssGetLastError**.

Remarks

The **RssParseFeed** function is an alternative to the **RssOpenFeed** function, enabling the application to process a news feed from alternative sources such as a database or compressed file. It is important to note that the string which contains the news feed XML must be properly formatted and conform to the RSS standard specification.

The data referenced in the **RSSCHANNEL** structure should be considered read-only and never modified by the application. Not all members of the structure may contain valid values, in which case those members will either have a value of zero or will specify NULL pointers. When the feed is closed, the members of this structure will no longer be valid, and therefore should never be stored by the application. If the application needs to store or modify this information, it should

create its own private copy of the data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrssv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RssCloseFeed](#), [RssInitialize](#), [RssOpenFeed](#), [RssRefreshFeed](#), [RssStoreFeed](#), [RssUninitialize](#), [RSSCHANNEL](#)

RssRefreshFeed Function

```
INT WINAPI RssRefreshFeed(  
    HCHANNEL hChannel,  
    LPRSSCHANNEL lpChannel  
);
```

The **RssRefreshFeed** function reloads the news feed and updates the items in the channel.

Parameters

hChannel

Handle to the news feed channel.

lpChannel

A pointer to an **RSSCHANNEL** structure which contains information about the news feed channel such as the feed title, hyperlink and description. If the parameter is not NULL, the structure is initialized by the function. If the parameter is NULL, it is ignored and no information is returned.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is **RSS_ERROR**. To get extended error information, call **RssGetLastError**.

Remarks

When the **RssRefreshFeed** function is called, the news feed is reloaded from the original source and the items in the channel are updated. For news feeds that are frequently updated, the *nTimeToLive* member of the **RSSCHANNEL** structure can provide a hint to the application as to how frequently the feed should be refreshed.

If the news feed was originally opened using an http or https URL, this function will download the updated feed from the server and store it temporarily on the local system. Accessing a remote feed requires that the application has permission to establish a connection with the server and will cause the application to block until the feed has been downloaded, the operation times out or an error occurs. The same timeout period and options will be used as when the feed was originally opened.

The **RssRefreshFeed** function should only be used if the feed was opened using the **RssOpenFeed** function, otherwise the function will fail with an error indicating that the operation is not supported.

The data referenced in the **RSSCHANNEL** structure should be considered read-only and never modified by the application. The members of this structure returned by previous calls to either the **RssOpenFeed** or **RssRefreshFeed** functions will no longer be valid and should not be referenced. Likewise, the members of an **RSSCHANNELITEM** structure will no longer be valid after this function returns.

It is important that the application does not make any assumptions about the number of news items in the channel, or the content associated with those items after the **RssRefreshFeed** function has been called. For example, never assume that the number of items in the channel remains the same, or that the item IDs for each item remains the same. If you need to find a specific item in the news feed, use the **RssFindItem** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrssv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RssCloseFeed](#), [RssFindItem](#), [RssGetItem](#), [RssOpenFeed](#), [RssStoreFeed](#), [RSSCHANNEL](#)

RssSetLastError Function

```
VOID WINAPI RssSetLastError(  
    DWORD dwErrorCode  
);
```

The **RssSetLastError** function sets the error code for the current thread. This function is typically used to clear the last error by passing a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or RSS_ERROR. Those functions which call **RssSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **RssGetLastError** function. The use of **RssGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csrsv10.lib

See Also

[RssGetErrorString](#), [RssGetLastError](#)

RssStoreFeed Function

```
INT WINAPI RssStoreFeed(  
    HCHANNEL hChannel,  
    LPCTSTR lpszFileName,  
    DWORD dwReserved  
);
```

The **RssStoreFeed** function stores the contents of the news feed in an XML formatted text file.

Parameters

hChannel

Handle to the news feed channel.

lpszFileName

A pointer to a string which specifies the name of the file on the local system. The contents of the news feed will be stored in this file. If the file does not exist, it will be created; otherwise it will overwrite the contents of the file.

dwReserved

A reserved parameter for future use. This value should always be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is **RSS_ERROR**. To get extended error information, call **RssGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csrsv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RssOpenFeed](#), [RssParseFeed](#)

RssUninitialize Function

```
VOID WINAPI RssUninitialize();
```

The **RssUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **RssInitialize** call before it can call any of the other the library functions. When it has completed the use of library, the application must call **RssUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There must be a call to **RssUninitialize** for every successful call to **RssInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrssv10.lib

See Also

[RssCloseFeed](#), [RssInitialize](#)

RssValidateFeed Function

```
INT WINAPI RssValidateFeed(  
    LPCTSTR lpszFeedUrl,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSYSTEMTIME lpstModified  
);
```

The **RssValidateFeed** function is used to validate a news feed, returning the number of items in the feed and the date it was last modified.

Parameters

lpszFeedUrl

A pointer to a string which specifies the URL for the news feed. To access a news feed on a web server, a standard http or https URL may be specified. To access a file on the local system or network share, a file name or file URL may be specified.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation. This parameter is ignored if the *lpszFeedUrl* parameter specifies a local file name or URL.

dwOptions

An unsigned integer that specifies one or more options. This parameter is reserved for future use and should always have a value of zero.

lpstModified

A pointer to a **SYSTEMTIME** structure which will specify the date that the feed was last modified when the function returns. If the parameter is NULL, it is ignored and no information is returned.

Return Value

If the function succeeds, the return value is the number of items in the news feed channel. If the function fails, the return value is `RSS_ERROR`. To get extended error information, call **RssGetLastError**.

Remarks

The **RssValidateFeed** function can be used to check that a news feed exists and is properly formatted. If the contents of the feed are valid, the function will return the number of items in the feed and the date that it was last modified. This can be useful for applications that want to periodically check a news feed and determine if the contents have changed.

The **SYSTEMTIME** structure that is populated by the function specifies the date when the feed was last modified. The function first checks the value of the `lastBuildDate` property of the feed channel. If that property is not defined, then it will use the value of the `pubDate` property. If neither are defined, then the structure members will have a value of zero.

The validation process imposes strict checks on the structure of the news feed and requires that it conform to the RSS specification. For example, the feed must have a title, link and description. Each item in the feed must have either a title or description, and the hyperlinks specified in the feed must be valid. If the feed XML is malformed, or a required property of the feed is invalid or missing, this function will fail.

If a news feed cannot be validated, it still may be possible to open the feed using the

RssOpenFeed function. By default, relaxed rules are used when parsing the contents of the feed and it does not check to ensure all required properties are defined and have valid values.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrsv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RssOpenFeed](#), [RssParseFeed](#), [RssRefreshFeed](#), [RssStoreFeed](#), [SYSTEMTIME](#)

News Feed Data Structures

- INITDATA
- RSSCHANNEL
- RSSCHANNELITEM
- SYSTEMTIME

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

INITDATA Structure

This structure contains information about the SocketTools library when the initialization function returns.

```
typedef struct _INITDATA
{
    DWORD        dwSize;
    DWORD        dwVersionMajor;
    DWORD        dwVersionMinor;
    DWORD        dwVersionBuild;
    DWORD        dwOptions;
    DWORD_PTR    dwReserved1;
    DWORD_PTR    dwReserved2;
    TCHAR        szDescription[128];
} INITDATA, *LPINITDATA;
```

Members

dwSize

Size of this structure. This structure member must be set to the size of the structure prior to calling the initialization function. Failure to do so will cause the function to fail.

dwVersionMajor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the major version number for the library.

dwVersionMinor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the minor version number for the library.

dwVersionBuild

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the build number for the library.

dwOptions

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved1

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved2

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

szDescription

A null terminated string which describes the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

RSSCHANNEL Structure

This structure contains information about the properties of the news feed channel.

```
typedef struct _RSSCHANNEL
{
    UINT        nItemCount;
    UINT        nTimeToLive;
    WORD        wVersionMajor;
    WORD        wVersionMinor;
    DWORD       dwFlags;
    DWORD       dwReserved;
    LPCTSTR     lpszTitle;
    LPCTSTR     lpszLink;
    LPCTSTR     lpszDescription;
    LPCTSTR     lpszCategory;
    LPCTSTR     lpszLanguage;
    LPCTSTR     lpszCopyright;
    LPCTSTR     lpszEditor;
    LPCTSTR     lpszWebmaster;
    LPCTSTR     lpszGenerator;
    LPCTSTR     lpszImageLink;
    LPCTSTR     lpszImageTitle;
    LPCTSTR     lpszImageUrl;
    SYSTEMTIME  stPublished;
    SYSTEMTIME  stLastBuild;
} RSSCHANNEL, *LPRSSCHANNEL;
```

Members

nItemCount

An integer value which specifies number of news items in the channel.

nTimeToLive

An integer value which specifies the frequency in seconds at which the feed should be refreshed to obtain updated information. Not all feeds specify a time-to-live, in which case this member will have a value of zero.

wVersionMajor

A word value which specifies the major version number for the news feed.

wVersionMinor

A word value which specifies the minor version number for the news feed.

dwFlags

A value which specifies one or more option flags for the news feed channel. Currently there are no option flags defined and this member is reserved for future expansion.

dwReserved

A value reserved for future expansion.

lpszTitle

A pointer to a string which specifies the name of the channel. If the content of the news feed corresponds to a website, this is typically the same as the title of the website. If a title has not been specified, this member will be NULL. Note that a strictly conforming news feed requires a title.

lpszLink

A pointer to a string which specifies a URL to the website corresponding to the channel. Note that this is not the URL of the news feed itself. Typically it is a link to the home page of the site which owns the news feed. If a link has not been specified, this member will be NULL. Note that a strictly conforming news feed requires a valid link URL.

lpszDescription

A pointer to a string which describes the channel. This provides an overview of the news feed and the type of information that is provided. If a description of the feed has not been specified, this member will be NULL. Note that a strictly conforming news feed requires a description.

lpszCategory

A pointer to a string which defines the category or categories that the channel belongs to. This property is optional and the category names themselves are user-defined. If a category has not been specified, this member will be NULL.

lpszLanguage

A pointer to a string which defines the language the channel is written in, using the standard language codes. This property is optional and if this member is NULL, the English language is typically presumed to be the default.

lpszCopyright

A pointer to a string which specifies a copyright notice for the content. If a copyright has not been specified, this member will be NULL.

lpszEditor

A pointer to a string which identifies the person responsible for managing the content of the news feed. If this property is defined, it is typically the name and email address of the feed editor. If an editor has not been specified, this member will be NULL.

lpszWebmaster

A pointer to a string which identifies the person responsible for technical issues related to the news feed. If this property is defined, it is typically the name and email address of a system administrator. If a webmaster has not been specified, this member will be NULL.

lpszGenerator

A pointer to a string which identifies the application that was used to create the news feed. If the application that generated the feed has not been specified, this member will be NULL.

lpszImageLink

A pointer to a string which specifies a URL to the website corresponding to the channel. In most cases, this is the same URL that is specified by the *lpszLink* member. If an image link has not been specified, this member will be NULL.

lpszImageTitle

A pointer to a string which identifies the image associated with the channel. This is usually a brief description of the image, and may be the same as the value specified by the *lpszTitle* member. If an image title has not been specified, this member will be NULL.

lpszImageUrl

A pointer to a string which specifies a URL for the image associated with the channel. An application can download this image and display it with the contents of the news feed. If an image URL has not been specified, this member will be NULL.

stPublished

The date that the news feed was published. For example, a feed that is associated with a weekly print publication may update this value once per week. Note that this is not necessarily the date

that the feed was last modified. If the channel does not specify the publish date, this structure will contain all zeroes.

stLastBuild

The date that the content of the channel was last modified. If the channel does not specify the build date, this structure will contain all zeroes.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

RSSCHANNELITEM Structure

This structure contains information about the properties of an item in a news feed channel.

```
typedef struct _RSSCHANNELITEM
{
    UINT            nItemId;
    DWORD           dwFlags;
    DWORD           dwReserved;
    LPCTSTR         lpszTitle;
    LPCTSTR         lpszLink;
    LPCTSTR         lpszText;
    LPCTSTR         lpszGuid;
    LPCTSTR         lpszAuthor;
    LPCTSTR         lpszSource;
    LPCTSTR         lpszComments;
    LPCTSTR         lpszEnclosure;
    SYSTEMTIME      stPublished;
} RSSCHANNELITEM, *LPRSSCHANNELITEM;
```

Members

nItemId

An integer which identifies this item in the channel.

dwFlags

A value which specifies one or more option flags for the item. Currently there are no option flags defined and this member is reserved for future expansion.

dwReserved

A value reserved for future expansion.

lpszTitle

A pointer to a string which specifies the title of the item. If a title for the item has not been specified, this member will be NULL.

lpszLink

A pointer to a string which specifies a URL that typically links to additional information related to the item. If a link for the item has not been specified, this member will be NULL.

lpszText

A pointer to a string which specifies a summary or description of the item. This may contain either plain text or HTML formatted text, and there is no fixed limit to the length of the text. If no text has been specified for the item, this member will be NULL.

lpszGuid

A pointer to a string which uniquely identifies the item in the channel. If this property is defined, it is guaranteed to be a unique, persistent value. It is important to note that this string does not have to be a standard GUID reference number, it can be any unique string. In many cases it is the same value as the item hyperlink specified by the *lpszLink* member, although an application should never depend on this behavior. If there is no unique identifier associated with the item, this member will be NULL.

lpszAuthor

A pointer to a string which identifies the author of the item. If this property is defined, it is typically the name and email address of the person who created the content that the item links to. If the author is not specified, this member will be NULL.

lpszSource

A pointer to a string which identifies the source of the item, specified as a URL for the original news feed that contained it. This typically used to propagate credit for items that are aggregated by a third-party and re-published in their own channel. If the source is not specified, this member will be NULL.

lpszComments

A pointer to a string which specifies a URL that links to further discussion about the item. Typically this is a link to the comment area of a weblog or a forum topic specific to the item. If a comment link is not specified, this member will be NULL.

lpszEnclosure

A pointer to a string which specifies a URL that links to a file related to the item. This is similar to an attachment in an email message, however instead of the item containing the contents of the attached file, it only specifies a link to the file. Enclosures are most commonly used with podcasting where an item is linked to an audio or video file, however the link may reference any type of file. If there is no enclosure specified for the item, this member will be NULL.

stPublished

The date that the item was published. If the item does not specify the publish date, this structure will contain all zeroes.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

SYSTEMTIME Structure

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include windows.h.

Post Office Protocol Library

List and retrieve email messages from a mail server.

Reference

- [Functions](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

File Name	CSPOPV10.DLL
Version	10.0.1468.2518
LibID	E4B30E06-4091-4A79-8A4C-17547C76A995
Import Library	CSPOPV10.LIB
Dependencies	None
Standards	RFC 1939

Overview

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

This library supports secure connections using the standard SSL and TLS protocols. Both implicit and explicit SSL connections can be established, enabling the library to work with a wide variety of servers.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location

on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Post Office Protocol Functions

Function	Description
PopAsyncConnect	Connect asynchronously to the specified server
PopAttachThread	Attach the specified client handle to another thread
PopCancel	Cancel the current blocking operation
PopChangePassword	Change the specified mail account password
PopCommand	Send a command to the server
PopConnect	Connect to the specified server
PopCreateSecurityCredentials	Allocate a structure to establish client security credentials
PopDeleteMessage	Delete the specified message from the mailbox
PopDeleteSecurityCredentials	Delete the specified client security credentials
PopDisableEvents	Disable the event notification mechanism
PopDisableTrace	Disable logging of socket function calls to the trace log
PopDisconnect	Disconnect from the current server
PopEnableEvents	Enable the client event notification mechanism
PopEnableTrace	Enable logging of socket function calls to a file
PopEventProc	Callback function that processes events generated by the client
PopFreezeEvents	Suspend and resume event handling by the client
PopGetErrorString	Return a description for the specified error code
PopGetHeaderValue	Return the value of the specified header field
PopGetLastError	Return the last error code
PopGetMessage	Retrieve the specified message from the server
PopGetMessageCount	Return the number of messages available in the mailbox
PopGetMessageCountEx	Return the number of messages available in the mailbox
PopGetMessageHeaders	Retrieve the specified message header from the server
PopGetMessageId	Return the message ID string for the specified message
PopGetMessageSender	Return the address of the message sender
PopGetMessageSize	Return the size of the specified message
PopGetMessageUid	Return the unique identifier for the specified message
PopGetMultiLine	Return the client multi-line output flag
PopGetResultCode	Return the result code from the previous command
PopGetResultString	Return the result string from the previous command
PopGetSecurityInformation	Return security information about the current client connection

PopGetStatus	Return the current status of the client
PopGetTimeout	Return the number of seconds until an operation times out
PopGetTransferStatus	Return data transfer statistics
PopInitialize	Initialize the library and validate the specified license key at runtime
PopIsBlocking	Determine if the client is blocked, waiting for information
PopIsConnected	Determine if the client is connected to the server
PopIsReadable	Determine if data can be read from the server
PopIsWritable	Determine if data can be written to the server
PopLogin	Login to the server
PopOpenMessage	Open the specified message for reading on the server
PopRead	Read data returned by the server
PopRegisterEvent	Register an event handler for the specified event
PopReset	Reset the client and return to a command state
PopSendMessage	Send a message through the mail server
PopSetLastError	Set the last error code
PopSetMultiLine	Set the client multi-line output flag
PopSetTimeout	Set the number of seconds until an operation times out
PopStoreMessage	Store the contents of a message in the specified file
PopUninitialize	Terminate use of the library by the application
PopWrite	Write data to the server

PopAsyncConnect Function

```
HCLIENT WINAPI PopAsyncConnect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSECURITYCREDENTIALS LpCredentials,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **PopAsyncConnect** function is used to establish a connection with the server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **PopConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 110. For secure connections, the default port number is 995. If the secure port number is specified, an implicit SSL/TLS connection will be established by default.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
POP_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
POP_OPTION_LINEBREAK	Message data that is received from the server is read as individual lines of text terminated by a carriage return and linefeed control sequence. This option can be useful for applications that need to use the lower level network I/O functions and must process the message text on a line-by-line basis. This option is not recommended for most

	<p>applications because it can have a negative impact on performance when retrieving large messages from the server.</p>
POP_OPTION_TUNNEL	<p>This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.</p>
POP_OPTION_TRUSTEDSITE	<p>This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.</p>
POP_OPTION_SECURE	<p>This option specifies that a secure connection should be established with the server and requires that the server support either the SSL or TLS protocol. This option is the same as specifying POP_OPTION_SECURE_EXPLICIT, which initiates the secure session using the STLS command.</p>
POP_OPTION_SECURE_EXPLICIT	<p>This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using either the SSL or TLS protocol and the STLS command.</p>
POP_OPTION_SECURE_IMPLICIT	<p>This option specifies the client should attempt to establish a secure connection with the server. It should only be used when the server expects an implicit SSL connection or does not implement RFC 2595 where the STLS command is used to negotiate a secure connection with the server.</p>
POP_OPTION_SECURE_FALLBACK	<p>This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.</p>
POP_OPTION_PREFER_IPV6	<p>This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.</p>

POP_OPTION_FREETHREAD

This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpCredentials

A pointer to a [SECURITYCREDENTIALS](#) structure which is used to establish the client credentials for a secure connection to the server. The function **PopCreateSecurityCredentials** can be used to create this structure if necessary. If a standard non-secure connection is being established, or client credentials are not required by the server, this parameter can be NULL.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous socket events that occur. If this parameter is NULL, a blocking connection is established with the server.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the *hEventWnd* parameter is NULL, this parameter must be specified as WM_NULL.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **PopGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
POP_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
POP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
POP_EVENT_READ	Data is available to read by the client. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the calling process is in asynchronous mode.
POP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
POP_EVENT_TIMEOUT	The client has timed out while waiting for a response from the server. Note that under some circumstances this event can be

	generated for a non-blocking connection, such as when the client is establishing a secure connection.
POP_EVENT_CANCEL	The client has canceled the current operation.
POP_EVENT_COMMAND	The client has processed a command that was sent to the server. The result code and result string can be used to determine if the response to the command. The high word of the IParam parameter should be checked, since this notification message will also be posed if the command cannot be executed.
POP_EVENT_PROGRESS	This event notification is sent periodically during lengthy blocking operations, such as retrieving a complete message from the server.

To cancel asynchronous notification and return the client to a blocking mode, use the **PopDisableEvents** function.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **PopAttachThread** function.

Specifying the POP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[PopConnect](#), [PopDisconnect](#), [PopInitialize](#), [PopLogin](#), [PopUninitialize](#)

PopAttachThread Function

```
DWORD WINAPI PopAttachThread(  
    HCLIENT hClient  
    DWORD dwThreadId  
);
```

The **PopAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **PopAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **PopAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **PopCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **PopUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: cspopv10.lib

See Also

[PopCancel](#), [PopAsyncConnect](#), [PopConnect](#), [PopDisconnect](#), [PopUninitialize](#)

PopCancel Function

```
INT WINAPI PopCancel(  
    HCLIENT hClient  
);
```

The **PopCancel** function cancels any outstanding blocking operation in the client, causing the blocking function to fail. The application may then retry the operation or terminate the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

When the **PopCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cspopv10.lib

See Also

[PopIsBlocking](#), [PopReset](#)

PopChangePassword Function

```
BOOL WINAPI PopChangePassword(  
    HCLIENT hClient,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszOldPassword,  
    LPCTSTR lpszNewPassword  
);
```

The **PopChangePassword** function changes the account password for the specified user.

Parameters

hClient

Handle to the client session.

lpszUserName

Pointer to a string which specifies the user name of the account who's password will be changed. It is not required that this be the same user name that was used to login to the mail server.

lpszOldPassword

Pointer to a string which specifies the current account password.

lpszNewPassword

Pointer to a string which specifies the new account password. When the function returns, the user's mailbox password will be set to this value.

Return Value

If the function succeeds, it will return a non-zero value. If the function fails, it will return zero. To get extended error information, call **PopGetLastError**.

Remarks

The **PopChangePassword** function is used to change the password associated with the specified account on the server. The function establishes a connection to a separate service running on the server, and does not use the POP3 protocol. For this function to succeed, the server must be configured to allow password changes using the "poppass" service, running on port 106.

Because passwords are sent over the network as clear text, this service is considered to be insecure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[PopConnect](#), [PopLogin](#)

PopCloseMessage Function

```
INT WINAPI PopCloseMessage(  
    HCLIENT hClient  
);
```

The **PopCloseMessage** function closes the current message that has been opened or created.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

If an message is being created, this function actually submits the message to the server. Note that the client application is responsible for generating the message headers as well as the body of the message. News messages conform to the same general characteristics of an email message.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cspopv10.lib

See Also

[PopOpenMessage](#), [PopRead](#)

PopCommand Function

```
BOOL WINAPI PopCommand(  
    HCLIENT hClient,  
    LPCTSTR lpszCommand,  
    LPCTSTR lpszParameter  
);
```

The **PopCommand** function sends a command to the server and returns the result code back to the caller. This function is typically used for site-specific commands not directly supported by the API.

Parameters

hClient

Handle to the client session.

lpszCommand

The command which will be executed by the server.

lpszParameter

An optional command parameter. If the command requires more than one parameter, then they should be combined into a single string, with a space separating each parameter. If the command does not accept any parameters, this value may be NULL.

Return Value

If the command was successful, the function returns a non-zero value. If the command failed, the function returns zero. To get extended error information, call **PopGetLastError**.

Remarks

A list of valid commands can be found in the technical specification for the protocol. Many servers will list supported commands when the HELP command is used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[PopGetMultiLine](#), [PopGetResultCode](#), [PopGetResultString](#), [PopSetMultiLine](#)

PopConnect Function

```
HCLIENT WINAPI PopConnect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSECURITYCREDENTIALS LpCredentials  
);
```

The **PopConnect** function is used to establish a connection with the server.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 110. For secure connections, the default port number is 995. If the secure port number is specified, an implicit SSL/TLS connection will be established by default.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
POP_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
POP_OPTION_LINEBREAK	Message data that is received from the server is read as individual lines of text terminated by a carriage return and linefeed control sequence. This option can be useful for applications that need to use the lower level network I/O functions and must process the message text on a line-by-line basis. This option is not recommended for most applications because it can have a negative impact on performance when retrieving large messages from the server.
POP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection

	and how the connection is established.
POP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
POP_OPTION_SECURE	This option specifies that a secure connection should be established with the server and requires that the server support either the SSL or TLS protocol. This option is the same as specifying POP_OPTION_SECURE_EXPLICIT, which initiates the secure session using the STLS command.
POP_OPTION_SECURE_EXPLICIT	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using either the SSL or TLS protocol and the STLS command.
POP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to establish a secure connection with the server. It should only be used when the server expects an implicit SSL connection or does not implement RFC 2595 where the STLS command is used to negotiate a secure connection with the server.
POP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
POP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
POP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpCredentials

A pointer to a [SECURITYCREDENTIALS](#) structure which is used to establish the client credentials for a secure connection to the server. The **PopCreateSecurityCredentials** function can be used

to create this structure if necessary. If a standard connection is being established, or client credentials are not required by the server, this parameter can be NULL.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is `INVALID_CLIENT`. To get extended error information, call **PopGetLastError**.

Remarks

To prevent this function from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **PopConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **PopAttachThread** function.

Specifying the `POP_OPTION_FREETHREAD` option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cspopv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[PopDisconnect](#), [PopInitialize](#), [PopLogin](#), [PopUninitialize](#)

PopCreateSecurityCredentials Function

```
BOOL WINAPI PopCreateSecurityCredentials(  
    DWORD dwProtocol,  
    DWORD dwOptions,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName,  
    LPVOID lpvReserved,  
    LPSECURITYCREDENTIALS* lppCredentials  
);
```

The **PopCreateSecurityCredentials** function creates a **SECURITYCREDENTIALS** structure.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is

	supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A pointer to a string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A pointer to a string which specifies the certificate owner's password. A value of NULL specifies

that no password is required. This parameter is only used if a PKCS12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A pointer to a string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

lpvReserved

Pointer reserved for future use. Set it to NULL when using this function.

lppCredentials

Pointer to an [LPSECURITYCREDENTIALS](#) pointer. The memory for the credentials structure will be allocated by this function and must be released by calling the **PopDeleteSecurityCredentials** function when it is no longer needed. The pointer value must be set to NULL before the function is called. It is important to note that this is a pointer to a pointer variable, not a pointer to the SECURITYCREDENTIALS structure itself.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **PopGetLastError**.

Remarks

The structure that is created by this function may be used as client credentials when establishing a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can

be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **PopAsyncConnect** or **PopConnect** functions.

Example

```
LPSECURITYCREDENTIALS lpSecCred = NULL;
PopCreateSecurityCredentials(SEcurity_PROTOCOL_DEFAULT,
                            0,
                            NULL,
                            NULL,
                            lpzCertStore,
                            lpzCertName,
                            NULL,
                            &lpSecCred);

hClient = PopConnect(lpzHostName,
                    POP_PORT_SECURE,
                    POP_TIMEOUT,
                    POP_OPTION_SECURE,
                    lpSecCred);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[PopAsyncConnect](#), [PopConnect](#), [PopDeleteSecurityCredentials](#), [PopGetSecurityInformation](#), [SECURITYCREDENTIALS](#)

PopDeleteMessage Function

```
INT WINAPI PopDeleteMessage(  
    HCLIENT hClient,  
    UINT nMessage  
);
```

The **PopDeleteMessage** function marks the specified message for deletion from the mailbox.

Parameters

hClient

Handle to the client session.

nMessage

Number of message to delete from the server. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

This function only marks the message for deletion. The message is not actually deleted until the user disconnects from the server. To prevent one or more marked messages from actually being deleted from the mailbox, call the **PopReset** function to reset the client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[PopGetMessage](#), [PopGetMessageCount](#), [PopReset](#)

PopDeleteSecurityCredentials Function

```
VOID WINAPI PopDeleteSecurityCredentials(  
    LPSECURITYCREDENTIALS* lppCredentials  
);
```

The **PopDeleteSecurityCredentials** function deletes an existing **SECURITYCREDENTIALS** structure.

Parameters

lppCredentials

Pointer to an **LPSECURITYCREDENTIALS** pointer. On exit from the function, the pointer will be NULL.

Return Value

None.

Example

```
if (lpSecCred)  
    PopDeleteSecurityCredentials(&lpSecCred);  
  
PopUninitialize();
```

Remarks

This function can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[PopCreateSecurityCredentials](#), [PopUninitialize](#)

PopDisableEvents Function

```
INT WINAPI PopDisableEvents(  
    HCLIENT hClient  
);
```

The **PopDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

This function affects both event notification and event callbacks. Any outstanding events in the message queue should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: cspopv10.lib

See Also

[PopEnableEvents](#), [PopFreezeEvents](#), [PopRegisterEvent](#)

PopDisableTrace Function

```
BOOL WINAPI PopDisableTrace();
```

The **PopDisableTrace** function disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

See Also

[PopEnableTrace](#)

PopDisconnect Function

```
INT WINAPI PopDisconnect(  
    HCLIENT hClient  
);
```

The **PopDisconnect** function terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

See Also

[PopAsyncConnect](#), [PopConnect](#), [PopUninitialize](#)

PopEnableEvents Function

```
INT WINAPI PopEnableEvents(  
    HCLIENT hClient,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **PopEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **PopRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the event notification window. This window receives a user-defined message which specifies the event that has occurred. If this value is NULL, event notification is disabled.

uEventMsg

An unsigned integer which specifies the user-defined message that is sent when an event occurs. This parameter's value must be greater than the value of WM_USER. If the *hEventWnd* parameter is NULL, this value must be specified as WM_NULL.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
POP_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
POP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
POP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.

POP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
POP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
POP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
POP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
POP_EVENT_PROGRESS	The client is in the process of sending or receiving data from the server. This event is called periodically during a transfer so that the client can update any user interface components such as a status control or progress bar.

As noted, some events are only generated when the client is asynchronous mode. These events depend on the Windows Sockets asynchronous notification mechanism.

If event notification is disabled by specifying a NULL window handle, there may still be outstanding events in the message queue that must be processed. Since event handling has been disabled, these events should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[PopDisableEvents](#), [PopFreezeEvents](#), [PopRegisterEvent](#)

PopEnableTrace Function

```
BOOL WINAPI PopEnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **PopEnableTrace** function enables the logging of Windows Sockets function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace function logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[PopDisableTrace](#)

PopEventProc Function

```
VOID CALLBACK PopEventProc(  
    HCLIENT hClient,  
    UINT nEvent,  
    DWORD dwError,  
    DWORD_PTR dwParam  
);
```

The **PopEventProc** function is an application-defined callback function that processes events generated by the client.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **PopRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **PopRegisterEvent** function. The **PopEventProc** function is a placeholder for the application-defined function name.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cspopv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[PopDisableEvents](#), [PopEnableEvents](#), [PopFreezeEvents](#), [PopRegisterEvent](#)

PopFreezeEvents Function

```
INT WINAPI PopFreezeEvents(  
    HCLIENT hClient,  
    BOOL bFreeze  
);
```

The **PopFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[PopDisableEvents](#), [PopEnableEvents](#), [PopRegisterEvent](#)

PopGetErrorString Function

```
INT WINAPI PopGetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);
```

The **PopGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **PopGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[PopGetLastError](#), [PopSetLastError](#)

PopGetHeaderValue Function

```
INT WINAPI PopGetHeaderValue(  
    HCLIENT hClient,  
    UINT nMessageId,  
    LPCTSTR LpszHeader,  
    LPTSTR LpszValue,  
    INT nMaxLength  
);
```

The **PopGetHeaderValue** function returns the value of a header field in the specified message.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve header value from. This value must be greater than zero. The first message in the mailbox is message number one.

lpszHeader

Pointer to a string which specifies the message header to retrieve. The colon should not be included in this string.

lpszValue

Pointer to a string buffer that will contain the value of the specified message header.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the function returns the length of the header field value. If the header field is not present in the message, the function will return a value of zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopGetHeaderValue** function returns the value of a header field from the specified message. This allows an application to be able to easily determine the value of a header (such as the sender, or the subject of the message) without downloading the entire header block or contents of the message.

This function uses the XTND XLST command, which is an extension to the POP3 protocol. Not all servers support the use of this command. If this command is not supported by the server, the function will attempt to retrieve the entire message header and return the value for the specified header field. This enables an application to use this function even if the server does not support command extensions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[PopGetMessageHeaders](#), [PopGetMessageId](#), [PopGetMessageSender](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

PopGetLastError Function

```
DWORD WINAPI PopGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **PopSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **PopGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **PopSetLastError(0)** when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or POP_ERROR. Those functions which call **PopSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[PopGetErrorString](#), [PopSetLastError](#)

PopGetMessage Function

```
INT WINAPI PopGetMessage(  
    HCLIENT hClient,  
    LONG nMessageId,  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwReserved  
);
```

The **PopGetMessage** function retrieves the specified message and copies the contents to a local buffer.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve from the server. This value must be greater than zero.

lpvBuffer

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvBuffer* parameter. If the *lpvBuffer* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the file that was downloaded.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopGetMessage** function is used to retrieve an message from the server and copy it into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the message. In this case, the *lpvBuffer* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvBuffer* parameter point to a global memory handle which will contain the message data when the function returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

This function will cause the current thread to block until the complete message has been retrieved, a timeout occurs or the operation is canceled. During the transfer, the POP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls.

Event notification must be enabled, either by calling **PopEnableEvents**, or by registering a callback function using the **PopRegisterEvent** function.

To determine the current status of a transfer while it is in progress, use the **PopGetTransferStatus** function.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Return the message into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter
nResult = PopGetMessage(hClient,
                       nMessageId,
                       &hgblBuffer,
                       &cbBuffer,
                       0);

if (nResult != POP_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // message text
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[PopEnableEvents](#), [PopGetMessageHeaders](#), [PopGetTransferStatus](#), [PopRegisterEvent](#)

PopGetMessageCount Function

```
INT WINAPI PopGetMessageCount(  
    HCLIENT hClient  
);
```

The **PopGetMessageCount** function returns the number of mail messages that are currently available in the mailbox.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, it returns the number of messages that are currently available. If no messages are available, either because the mailbox is empty or all of the messages have been deleted, this function will return zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cspopv10.lib

See Also

[PopDeleteMessage](#), [PopGetHeaderValue](#), [PopGetMessage](#), [PopGetMessageCountEx](#), [PopGetMessageHeaders](#), [PopStoreMessage](#)

PopGetMessageCountEx Function

```
INT WINAPI PopGetMessageCountEx(  
    HCLIENT hClient,  
    UINT *LpnLastMessage,  
    DWORD *LpdwMailboxSize  
);
```

The **PopGetMessageCountEx** function returns the number of mail messages that are currently available in the mailbox.

Parameters

hClient

Handle to the client session.

lpnLastMessage

Address of a variable that receives the number of the last valid message in the mailbox. If a NULL value is specified, this argument is ignored.

lpdwMailboxSize

Address of a variable that receives the current size of the mailbox. This value will decrease as messages are deleted. If a NULL value is specified, this argument is ignored.

Return Value

If the function succeeds, it returns the number of messages that are currently available. If no messages are available, either because the mailbox is empty or all of the messages have been deleted, this function will return zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopGetMessageCountEx** function returns the number of messages available in the mailbox, the last valid message number in the mailbox and the current size of the mailbox in bytes.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csppv10.lib

See Also

[PopDeleteMessage](#), [PopGetHeaderValue](#), [PopGetMessage](#), [PopGetMessageHeaders](#), [PopStoreMessage](#)

PopGetMessageHeaders Function

```
INT WINAPI PopGetMessageHeaders(  
    HCLIENT hClient,  
    LONG nMessageId,  
    LPVOID lpvHeaders,  
    LPDWORD lpdwLength  
);
```

The **PopGetMessageHeaders** function retrieves the headers for the specified message from the server.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve from the server. This value must be greater than zero.

lpvHeaders

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvHeaders* parameter. If the *lpvHeaders* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the message that was downloaded.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopGetMessageHeaders** function is used to retrieve an message header block from the server and copy it into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the file. In this case, the *lpvHeaders* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvHeaders* parameter point to a global memory handle which will contain the message headers when the function returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur.

This function will cause the current thread to block until the transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the POP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **PopEnableEvents**, or by registering a callback function using the **PopRegisterEvent** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[PopGetHeaderValue](#), [PopGetMessage](#), [PopGetMessageId](#), [PopOpenMessage](#)

PopGetMessageId Function

```
INT WINAPI PopGetMessageId(  
    HCLIENT hClient,  
    UINT nMessageId,  
    LPTSTR lpszMessageId,  
    INT nMaxLength  
);
```

The **PopGetMessageId** function returns the message identifier for the specified message.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve the unique identifier for. This value must be greater than zero. The first message in the mailbox is message number one.

lpszMessageId

Address of a string buffer to receive the message identifier. This should be at least 64 bytes in length.

nMaxLength

The maximum length of the string buffer.

Return Value

If the function succeeds, the return value is the length of the unique identifier string. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopGetMessageId** function returns the message identifier from the Message-ID header of the specified message. The returned value is typically a string which specifies the domain, date and timestamp for the message that is created when the message is submitted to the mail server for delivery. To obtain a unique identifier for the message in the mailbox, it is recommended that you use the **PopGetMessageUid** function instead.

This function uses the XTND XLST command to obtain the value of the "Message-ID" header field. If this command is not supported by the server, the function will attempt to retrieve the entire message header and return the value for the specified header field. This enables an application to use this function even if the server does not support command extensions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[PopGetHeaderValue](#), [PopGetMessage](#), [PopGetMessageHeaders](#), [PopGetMessageSender](#), [PopGetMessageUid](#)

PopGetMessageSender Function

```
INT WINAPI PopGetMessageSender(  
    HCLIENT hClient,  
    UINT nMessageId,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

The **PopGetMessageSender** function returns the sender's address for the specified message.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve header value from. This value must be greater than zero. The first message in the mailbox is message number one.

lpszAddress

Pointer to a string buffer that will contain the address of the message sender.

nMaxLength

The maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the function returns the length of the address. If the sender cannot be determined, the function will return a value of zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopGetMessageSender** function returns the email address of the user who sent the specified message. This allows an application to be able to easily determine the sender, without downloading the entire header block or contents of the message.

This function uses the XSENDER command, which is an extension to the POP3 protocol, to determine the address of the authenticated sender of the message. If the command is not supported, or the server was unable to authenticate the sender, the function will use the XTND XLST command to obtain the value of the "From" header field. If this command is not supported by the server, the function will attempt to retrieve the entire message header and return the value for the specified header field. This enables an application to use this function even if the server does not support command extensions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[PopGetHeaderValue](#), [PopGetMessageHeaders](#), [PopGetMessageId](#), [PopGetMessageUid](#)

PopGetMessageSize Function

```
DWORD WINAPI PopGetMessageSize(  
    HCLIENT hClient,  
    UINT nMessageId  
);
```

The **PopGetMessageSize** function returns the size of the specified message.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve size of. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

If the function succeeds, the return value is the size of the specified message in bytes. If the function fails, the return value is POP_ERROR. To get extended error information, call

PopGetLastError.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cspopv10.lib

See Also

[PopGetHeaderValue](#), [PopGetMessageHeaders](#), [PopGetMessageId](#), [PopGetMessageSender](#), [PopGetTransferStatus](#)

PopGetMessageUid Function

```
INT WINAPI PopGetMessageUid(  
    HCLIENT hClient,  
    UINT nMessageId,  
    LPTSTR lpszMessageUID,  
    INT nMaxLength  
);
```

The **PopGetMessageUid** function returns the unique identifier (UID) for the specified message in the current mailbox.

Parameters

hClient

Handle to the client session.

nMessageId

Number of message to retrieve the unique identifier for. This value must be greater than zero. The first message in the mailbox is message number one.

lpszMessageUID

Address of a string buffer to receive the unique identifier for the specified message. This should be at least 64 bytes in length.

nMaxLength

The maximum length of the string buffer for the message UID.

Return Value

If the function succeeds, it returns a non-zero value. If no unique identifier is assigned to the message, the function will return zero. If an error occurs, the function returns POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopGetMessageUid** function returns the unique message identifier for the specified message. The returned value is a string which can be used to uniquely identify a specific message in the mailbox across multiple client sessions. This is commonly used by mail clients to determine if they have already retrieved a message from the server in a previous session. The UID can also be used as a key or component of the file name to reference the message after it has been stored on the local system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cspopv10.lib

See Also

[PopGetHeaderValue](#), [PopGetMessage](#), [PopGetMessageHeaders](#), [PopGetMessageId](#), [PopGetMessageSender](#)

PopGetMultiLine Function

```
INT WINAPI PopGetMultiLine(  
    HCLIENT hClient,  
    LPBOOL lpbMultiLine  
);
```

The **PopGetMultiLine** function returns the value of the client multi-line flag in the specified boolean parameter.

Parameters

hClient

Handle to the client session.

lpbMultiLine

A pointer to a boolean variable. This variable will be set to the current value of the client's internal multi-line flag.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The multi-line flag is used by the library to determine if multiple lines of data will be returned by the server as the result of a command. Unlike a single line response, which consists of a result code and result string, a multi-line response consists of one or more lines of text, terminated by a special end-of-data marker.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[PopCommand](#), [PopGetResultCode](#), [PopGetResultString](#), [PopSetMultiLine](#)

PopGetResultCode Function

```
BOOL WINAPI PopGetResultCode(  
    HCLIENT hClient  
);
```

The **PopGetResultCode** function reads the result code returned by the server in response to a command. The result code is a Boolean value, and indicates if the operation succeeded or failed.

Parameters

hClient

Handle to the client session.

Return Value

If the previous command was successful, the function returns a non-zero value. If the previous command failed, the function returns zero. To get extended error information, call **PopGetLastError**.

Remarks

Unlike most other Internet application protocols, the Post Office Protocol does not return numeric result codes to indicate success or failure. If a command is successful, the server will respond with the string "+OK" and this is indicated by the **PopGetResultCode** function returning a non-zero value. If the command fails, the server will respond with the string "-ERR" along with a description of the error, and this is indicated by the function returning a value of zero. The description of the error returned by the server can be obtained by calling the **PopGetResultString** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

See Also

[PopCommand](#), [PopGetResultString](#)

PopGetResultString Function

```
INT WINAPI PopGetResultString(  
    HCLIENT hClient,  
    LPTSTR lpszResult,  
    INT nMaxLength  
);
```

The **PopGetResultString** function returns the last message sent by the server along with the result code.

Parameters

hClient

Handle to the client session.

lpszResult

A pointer to the buffer that will contain the result string returned by the server.

nMaxLength

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopGetResultString** function is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **PopGetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[PopCommand](#), [PopGetResultCode](#)

PopGetSecurityInformation Function

```
BOOL WINAPI PopGetSecurityInformation(  
    HCLIENT hClient,  
    LPSECURITYINFO lpSecurityInfo  
);
```

The **PopGetSecurityInformation** function returns security protocol, encryption and certificate information about the current client connection.

Parameters

hClient

Handle to the client session.

lpSecurityInfo

A pointer to a [SECURITYINFO](#) structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **PopGetLastError**.

Remarks

This function is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value `SECURITY_PROTOCOL_NONE`.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;  
  
securityInfo.dwSize = sizeof(SECURITYINFO);  
if (PopGetSecurityInformation(hClient, &securityInfo))  
{  
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)  
    {  
        MessageBox(NULL, _T("The connection is not secure"),  
                    _T("Connection"), MB_OK);  
    }  
    else  
    {  
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)  
        {  
            MessageBox(NULL, _T("The connection is secure"),  
                        _T("Connection"), MB_OK);  
        }  
        else  
        {  
            MessageBox(NULL, _T("The server certificate not valid"),  
                        _T("Connection"), MB_OK);  
        }  
    }  
}
```

```
}  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[PopConnect](#), [PopDisconnect](#), [SECURITYINFO](#)

PopGetStatus Function

```
INT WINAPI PopGetStatus(  
    HCLIENT hClient  
);
```

The **PopGetStatus** function returns the current status of the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopGetStatus** function returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Constant	Description
1	POP_STATUS_IDLE	The client is current idle and not sending or receiving data.
2	POP_STATUS_CONNECT	The client is establishing a connection with the server.
3	POP_STATUS_READ	The client is reading data from the server.
4	POP_STATUS_WRITE	The client is writing data to the server.
5	POP_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

See Also

[PopIsBlocking](#), [PopIsConnected](#), [PopIsReadable](#), [PopIsWritable](#)

PopGetTimeout Function

```
INT WINAPI PopGetTimeout(  
    HCLIENT hClient  
);
```

The **PopGetTimeout** function returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

See Also

[PopConnect](#), [PopIsReadable](#), [PopIsWritable](#), [PopRead](#), [PopSetTimeout](#)

PopGetTransferStatus Function

```
INT WINAPI PopGetTransferStatus(  
    HCLIENT hClient,  
    LPPOPTRANSFERSTATUS lpStatus  
);
```

The **PopGetTransferStatus** function returns information about the current file transfer in progress.

Parameters

hClient

Handle to the client session.

lpStatus

A pointer to an [POPTRANSFERSTATUS](#) structure which contains information about the status of the current file transfer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopGetTransferStatus** function returns information about the current file transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no file currently being transferred, this function will return the status of the last successful transfer made by the client.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cspopv10.lib

See Also

[PopEnableEvents](#), [PopGetStatus](#), [PopRegisterEvent](#)

PopInitialize Function

```
BOOL WINAPI PopInitialize(  
    LPCTSTR lpszLicenseKey,  
    LPINITDATA lpData  
);
```

The **PopInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpData

Pointer to an INITDATA data structure. This parameter may be NULL if the initialization data for the library is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **PopGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must insure that this function is called with a valid license key.

If the *lpData* argument is specified, it must point to an INITDATA structure which has been initialized by setting the *dwSize* member to the size of the structure. All other structure members should be set to zero. If the function is successful, then the INITDATA structure will be filled with identifying information about the library.

Although it is only required that **PopInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **PopUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **PopInitialize** or **PopUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

PopIsBlocking Function

```
BOOL WINAPI PopIsBlocking(  
    HCLIENT hClient  
);
```

The **PopIsBlocking** function is used to determine if the client is currently performing a blocking operation.

Parameters

hClient

Handle to the client session.

Return Value

If the client is performing a blocking operation, the function returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the function returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking function may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **PopIsBlocking** function can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cspopv10.lib`

See Also

[PopCancel](#)

PopIsConnected Function

```
BOOL WINAPI PopIsConnected(  
    HCLIENT hClient  
);
```

The **PopIsConnected** function is used to determine if the client is currently connected to a server.

Parameters

hClient

Handle to the client session.

Return Value

If the client is connected to a server, the function returns a non-zero value. If the client is not connected, or the client handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[PopIsBlocking](#), [PopIsReadable](#), [PopIsWritable](#)

PopIsReadable Function

```
BOOL WINAPI PopIsReadable(  
    HCLIENT hClient,  
    INT nTimeout,  
    LPDWORD lpdwAvail  
);
```

The **PopIsReadable** function is used to determine if data is available to be read from the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the function returns a non-zero value. If the client cannot read any data, the function returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

See Also

[PopGetStatus](#), [PopIsBlocking](#), [PopIsConnected](#), [PopIsWritable](#), [PopRead](#)

PopIsWritable Function

```
BOOL WINAPI PopIsWritable(  
    HCLIENT hClient,  
    INT nTimeout  
);
```

The **PopIsWritable** function is used to determine if data can be written to the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the function returns a non-zero value. If the client cannot write any data, the function returns zero.

Remarks

Although this function can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cspopv10.lib`

See Also

[PopGetStatus](#), [PopIsBlocking](#), [PopIsConnected](#), [PopIsReadable](#), [PopWrite](#)

PopLogin Function

```
INT WINAPI PopLogin(  
    HCLIENT hClient,  
    UINT nAuthType,  
    LPCTSTR LpszUserName,  
    LPCTSTR LpszPassword  
);
```

The **PopLogin** function authenticates the specified user in on the server. This function must be called after the connection has been established, and before attempting to retrieve messages or perform any other function on the server.

Parameters

hClient

Handle to the client session.

nAuthType

Identifies the type of authentication that should be used when the client logs in to the mail server. The following authentication methods are supported:

Constant	Description
POP_AUTH_DEFAULT	The default authentication scheme which sends the username and password as cleartext to the server. Because the user credentials are not encrypted, this method should only be used over a secure connection. This is the same as specifying POP_AUTH_PASS as the authentication method.
POP_AUTH_PASS	The username and password is sent to the server using the USER and PASS commands. This authentication method is supported by most servers and is the default authentication type. The credentials are not encrypted and this method should only be used over secure connections.
POP_AUTH_APOP	The APOP authentication method which uses an MD5 digest of the password. This method has been deprecated is not supported by all servers. It should only be used if required by legacy mail servers which do not support the SASL authentication methods.
POP_AUTH_LOGIN	This authentication type will use the LOGIN method to authenticate the client session. This encodes the username and password in a specific format, but the credentials are not encrypted and should only be used over a secure connection. The server must support the Simple Authentication and Security Layer (SASL) mechanism as defined in RFC 4422.
POP_AUTH_PLAIN	This authentication type will use the PLAIN method to authenticate the client session. This encodes the username and password in a specific format, but the credentials are not encrypted and should only be used over a secure connection. The server must support the PLAIN Simple Authentication and Security Layer (SASL) mechanism as defined in RFC 4616.

POP_AUTH_XOAUTH2	This authentication type will use the XOAUTH2 method to authenticate the client session. This authentication method does not require the user password, instead the <i>lpszPassword</i> parameter must specify the OAuth 2.0 bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this function will fail.
POP_AUTH_BEARER	This authentication type will use the OAUTHBEARER method to authenticate the client session as defined in RFC 7628. This authentication method does not require the user password, instead the <i>lpszPassword</i> parameter must specify the OAuth 2.0 bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this function will fail.

lpszUserName

A null terminated string which specifies the user name to be used to authenticate the current client session. For many service providers, the user name is the full email address of the user which owns the mailbox. In some cases, this may only be the portion of their email address before the domain name.

lpszPassword

A null terminated string which specifies the password to be used when authenticating the current client session. If you are using the POP_AUTH_XOAUTH2 or POP_AUTH_BEARER authentication methods, this parameter is not a password, instead it specifies the OAuth 2.0 bearer token provided by the mail service.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The POP_AUTH_LOGIN and POP_AUTH_PLAIN authentication methods require the mail server support the Simple Authentication and Security Layer (SASL) AUTH command as defined in RFC 5034. Most modern mail servers do support one or both of these methods, and they are generally preferred over the POP_AUTH_PASS method when possible. However, for backwards compatibility with legacy servers, the API will default to using POP_AUTH_PASS for client authentication.

You should only use an OAuth 2.0 authentication method if you understand the process of how to request the access token. Obtaining an access token requires registering your application with the mail service provider (e.g.: Microsoft or Google), getting a unique client ID associated with your application and then requesting the access token using the appropriate scope for the service. Obtaining the initial token will typically involve interactive confirmation on the part of the user, requiring they grant permission to your application to access their mail account.

The POP_AUTH_XOAUTH2 and POP_AUTH_BEARER authentication methods are similar, but they are not interchangeable. Both use an OAuth 2.0 bearer token to authenticate the client session, but they differ in how the token is presented to the server. It is currently preferable to use the XOAUTH2 method because it is more widely available and some service providers do not yet support the OAUTHBEARER method.

Your application should not store an OAuth 2.0 bearer token for later use. They have a relatively short lifespan, typically about an hour, and are designed to be used with that session. You should

specify offline access as part of the OAuth 2.0 scope if necessary and store the refresh token provided by the service. The refresh token has a much longer validity period and can be used to obtain a new bearer token when needed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[PopAsyncConnect](#), [PopConnect](#), [PopGetMessage](#), [PopGetMessageCountEx](#), [PopInitialize](#)

PopOpenMessage Function

```
INT WINAPI PopOpenMessage(  
    HCLIENT hClient,  
    UINT nMessageId,  
    DWORD dwReserved  
);
```

The **PopOpenMessage** function opens the specified message for reading.

Parameters

hClient

Handle to the client session.

nMessageId

Number that specifies which message to open. This value must be greater than zero. The first message in the mailbox is message one.

dwReserved

A reserved parameter. This value must be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopOpenMessage** function is used to begin the process of reading the contents of a message from the server. Similar to how a file is opened and read, this method is followed by one or more calls to the **PopRead** method. When the entire contents of the message has been read, the **PopCloseMessage** method is used to close the message, completing the transaction on the server.

This is a lower-level function which enables the application to process the message as the contents are being returned by the server. In general, it is recommended that most applications use the **PopGetMessage** method instead, which provides a simpler method for retrieving the contents of a message.

It is important to note that you cannot use this function to read the partial contents of a message. Opening a message on the server begins a process where the entire message contents must be read and the message closed before the next command can be issued to the server. If you only want to obtain the headers for a message, use the **PopGetMessageHeaders** function instead.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

See Also

[PopCloseMessage](#), [PopGetMessage](#), [PopGetMessageHeaders](#), [PopIsReadable](#), [PopRead](#)

PopRead Function

```
INT WINAPI PopRead(  
    HCLIENT hClient,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **PopRead** function reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hClient

Handle to the client session.

lpBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that there is no more data available to be read. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

When **PopRead** is called and the client is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[PopGetMessage](#), [PopGetMessageHeaders](#), [PopIsBlocking](#), [PopIsReadable](#), [PopOpenMessage](#)

PopRegisterEvent Function

```
INT WINAPI PopRegisterEvent(  
    HCLIENT hClient,  
    UINT nEvent,  
    POPEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

The **PopRegisterEvent** function registers an event handler for the specified event.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Constant	Description
POP_EVENT_CONNECT	The connection to the server has completed.
POP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
POP_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
POP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
POP_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
POP_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.
POP_EVENT_COMMAND	A command has been issued by the client and the server response has been received and processed. This event can be used to log the result codes and messages returned by the server in response to actions taken by the client.
POP_EVENT_PROGRESS	The client is in the process of sending or receiving data from the server. This event is called periodically during a

	transfer so that the client can update any user interface components such as a status control or progress bar.
--	--

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **PopEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopRegisterEvent** function associates a callback function with a specific event. The event handler is an **PopEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This function is typically used to register an event handler that is invoked while a message is being retrieved. The POP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one POP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

See Also

[PopDisableEvents](#), [PopEnableEvents](#), [PopEventProc](#), [PopFreezeEvents](#)

PopReset Function

```
BOOL WINAPI PopReset(  
    HCLIENT hClient  
);
```

The **PopReset** function resets the client state and resynchronizes with the server. This function is typically called after an unexpected error has occurred, or an operation has been canceled.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **PopGetLastError**.

Remarks

This function will prevent any messages marked for deletion from actually being deleted from the mailbox. The client cannot be reset while the client is in a blocked state.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cspopv10.lib

See Also

[PopCancel](#), [PopDeleteMessage](#), [PopIsBlocking](#)

PopSendMessage Function

```
INT WINAPI PopSendMessage(  
    HCLIENT hClient,  
    LPVOID lpMessage,  
    DWORD dwMessageSize,  
    DWORD dwReserved  
);
```

The **PopSendMessage** function sends a message to the specified recipients.

Parameters

hClient

Handle to the client session.

lpMessage

Pointer to a buffer which contains the message to be submitted to the mail server for delivery.

dwMessageSize

The length of the buffer in bytes. This specifies the number of bytes to be written to the mail server. This value must be greater than zero.

dwReserved

A reserved parameter. This value must be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopSendMessage** function sends a message through the POP3 server using the XTND XMIT command. The specified file must be in the standard format as described in RFC 822, with the recipient addresses specified in the To: and Cc: header fields. Some servers may support blind carbon copies by using addresses specified in a Bcc: header field, and then removing those addresses from the header before delivering the message.

Note that not all POP3 servers support this command, and it is recommended that you use the Simple Mail Transfer Protocol (SMTP) for general mail delivery purposes.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[PopGetMessage](#), [PopStoreMessage](#)

PopSetLastError Function

```
VOID WINAPI PopSetLastError(  
    DWORD dwErrorCode  
);
```

The **PopSetLastError** function sets the last error code for the current thread. This function is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or POP_ERROR. Those functions which call **PopSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **PopGetLastError** function. The use of **PopGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

See Also

[PopGetErrorString](#), [PopGetLastError](#)

PopSetMultiLine Function

```
INT WINAPI PopSetMultiLine(  
    HCLIENT hClient,  
    BOOL bMultiLine  
);
```

The **PopSetMultiLine** function sets the client multi-line flag into the specified value.

Parameters

hClient

Handle to the client session.

bMultiLine

A boolean flag which determines if the client is processing multiple lines of data as the result of a command.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The multi-line flag is used by the library to determine if multiple lines of data will be returned by the server as the result of a command. Unlike a single line response, which consists of a result code and result string, a multi-line response consists of one or more lines of text, terminated by a special end-of-data marker.

The **PopSetMultiLine** function should only be used in conjunction with the **PopCommand** function. If a command is issued which would result in multiple lines of output, the multi-line flag must be set TRUE. The multi-line flag must be set after each command, since it is reset to FALSE with each command that is sent to the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[PopCommand](#), [PopGetMultiLine](#), [PopGetResultCode](#), [PopGetResultString](#)

PopSetTimeout Function

```
INT WINAPI PopSetTimeout(  
    HCLIENT hClient,  
    INT nTimeout  
);
```

The **PopSetTimeout** function sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

See Also

[PopConnect](#), [PopGetTimeout](#), [PopRead](#)

PopStoreMessage Function

```
INT WINAPI PopStoreMessage(  
    HCLIENT hClient,  
    UINT nMessageId,  
    LPCTSTR lpszFileName  
);
```

The **PopStoreMessage** function stores a message in the specified file.

Parameters

hClient

Handle to the client session.

nMessageId

Number of the message to retrieve. This value must be greater than zero. The first message in the mailbox is message number one.

lpszFileName

Pointer to a string which specifies the file that the message will be stored in. If an empty string or NULL pointer is passed as an argument, the message is copied to the system clipboard.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The **PopStoreMessage** function provides a method of retrieving and storing a message on the local system. The contents of the message is stored as a text file, using the specified file name. This function always causes the caller to block until the entire message has been retrieved, even if the client has been put in asynchronous mode.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[PopGetMessage](#), [PopGetMessageHeaders](#), [PopGetTransferStatus](#), [PopSendMessage](#)

PopUninitialize Function

```
VOID WINAPI PopUninitialize();
```

The **PopUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **PopInitialize** call before it can call any of the other library functions. When it has completed the use of library, the application must call **PopUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There must be a call to **PopUninitialize** for every successful call to **PopInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cspopv10.lib

See Also

[PopDisconnect](#), [PopInitialize](#)

PopWrite Function

```
INT WINAPI PopWrite(  
    HCLIENT hClient,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **PopWrite** function sends the specified number of bytes to the server.

Parameters

hClient

Handle to the client session.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is POP_ERROR. To get extended error information, call **PopGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the POP_EVENT_WRITE asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cspopv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[PopIsBlocking](#), [PopIsWritable](#), [PopRead](#), [PopSendMessage](#)

Post Office Protocol Data Structures

- INITDATA
- POPTRANSFERSTATUS
- SECURITYCREDENTIALS
- SECURITYINFO
- SYSTEMTIME

INITDATA Structure

This structure contains information about the SocketTools library when the initialization function returns.

```
typedef struct _INITDATA
{
    DWORD      dwSize;
    DWORD      dwVersionMajor;
    DWORD      dwVersionMinor;
    DWORD      dwVersionBuild;
    DWORD      dwOptions;
    DWORD_PTR  dwReserved1;
    DWORD_PTR  dwReserved2;
    TCHAR      szDescription[128];
} INITDATA, *LPINITDATA;
```

Members

dwSize

Size of this structure. This structure member must be set to the size of the structure prior to calling the initialization function. Failure to do so will cause the function to fail.

dwVersionMajor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the major version number for the library.

dwVersionMinor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the minor version number for the library.

dwVersionBuild

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the build number for the library.

dwOptions

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved1

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved2

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

szDescription

A null terminated string which describes the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

POPTRANSFERSTATUS Structure

This structure is used by the [PopGetTransferStatus](#) function to return information about a message transfer in progress.

```
typedef struct _POPTRANSFERSTATUS
{
    UINT    nMessageId;
    DWORD   dwBytesTotal;
    DWORD   dwBytesCopied;
    DWORD   dwBytesPerSecond;
    DWORD   dwTimeElapsed;
    DWORD   dwTimeEstimated;
} POPTRANSFERSTATUS, *LPPOPTRANSFERSTATUS;
```

Members

nMessageId

The message ID of the current message that is being transferred.

dwBytesTotal

The total number of bytes that will be transferred. If the message is being copied from the server to the local host, this is the size of the message on the server. If the message is being posted to the server, it is the size of message on the local system. If the message size cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the transfer started.

dwTimeEstimated

The estimated number of seconds until the transfer is completed. This is based on the average number of bytes transferred per second.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

SECURITYCREDENTIALS Structure

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on

	what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that

will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A pointer to a null terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A pointer to a null terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A pointer to a null terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A pointer to a null terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
------------	-------------

CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
----	---

MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
----	--

ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.
------	--

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

A pointer to a null terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used for SSH connections

and should always be NULL when establishing a secure connection using SSL or TLS.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* member should be set to CREDENTIAL_STORE_FILENAME and the *lpzCertStore* member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS#12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the *lpzPassword* member or the library will be unable to access the certificate.

Note that the *lpzUserName* and *lpzPassword* members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

The TLS 1.1 and TLS 1.2 protocols are only supported on Windows 7, Windows Server 2008 R2 and later versions of the platform. If these options are specified and the application is running on Windows XP or Windows Vista, the protocol version will be downgraded to TLS 1.0 for backwards compatibility with those platforms.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

SECURITYINFO Structure

This structure contains information about a secure connection that has been established.

```
typedef struct _SECURITYINFO
{
    DWORD        dwSize;
    DWORD        dwProtocol;
    DWORD        dwCipher;
    DWORD        dwCipherStrength;
    DWORD        dwHash;
    DWORD        dwHashStrength;
    DWORD        dwKeyExchange;
    DWORD        dwCertStatus;
    SYSTEMTIME   stCertIssued;
    SYSTEMTIME   stCertExpires;
    LPCTSTR      lpszCertIssuer;
    LPCTSTR      lpszCertSubject;
    LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to **sizeof(SECURITYINFO)** prior to passing the address of this structure to the function. Note that if this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the protocol that was selected to establish the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The

	<p>correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.</p>
SECURITY_PROTOCOL_TLS10	<p>The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS11	<p>The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS12	<p>The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.</p>

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_CIPHER_RC2	The RC2 block cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
SECURITY_CIPHER_DES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher, using 56-bit

	keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher (also known as the Rijndael cipher) is a fixed block size cipher which use a key size of 128, 192 or 256 bits. This cipher is supported on Windows XP SP3 and later versions of the operating system.
SECURITY_CIPHER_SKIPJACK	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
SECURITY_CIPHER_BLOWFISH	The Blowfish block cipher was selected. This is a variable key length cipher up to 448 bits, using a 64-bit block size.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected. One of the following values will be returned:

Constant	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected. One of the following values will be returned:

--	--

Constant	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography. This key exchange algorithm is only supported on Windows XP SP3 and later versions of the operating system.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server. This member only has meaning for connections using the SSL or TLS protocols. One of the following values will be returned:

Constant	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertIssuer

A pointer to a string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertSubject

A pointer to a string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszFingerprint

A pointer to a string which contains a sequence of hexadecimal values that uniquely identify the server. This member is only used when a connection has been established using the Secure Shell (SSH) protocol.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Unicode: Implemented as Unicode and ANSI versions.

SYSTEMTIME Structure

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include windows.h.

Remote Command Protocol Library

Execute commands on a server or establish an interactive terminal session.

Reference

- [Functions](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

File Name	CSRSHV10.DLL
Version	10.0.1468.2518
LibID	4893754D-AF7A-4892-9320-D829EEC98DD6
Import Library	CSRSHV10.LIB
Dependencies	None
Standards	RFC 1282

Overview

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

This library should not be used when connecting to a server over the Internet because the user credentials are sent as unencrypted text. For secure remote command execution and interactive terminal sessions, it is recommended that you use the SocketTools Secure Shell (SSH) library instead.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Remote Command Protocol Functions

Function	Description
RshAsyncExecute	Execute a command on the specified server
RshAsyncLogin	Establish an asynchronous login session with the specified server
RshAttachThread	Attach the specified client handle to another thread
RshCancel	Cancel the current blocking operation
RshDisableEvents	Disable asynchronous event notification
RshDisableTrace	Disable logging of socket function calls to the trace log
RshDisconnect	Disconnect from the current server
RshEnableEvents	Enable asynchronous event notification
RshEnableTrace	Enable logging of socket function calls to a file
RshEventProc	Callback function that processes events generated by the client
RshExecute	Execute a command on the server
RshFreezeEvents	Suspend asynchronous event processing
RshGetErrorString	Return a description for the specified error code
RshGetLastError	Return the last error code
RshGetStatus	Return the current client status
RshGetTimeout	Return the number of seconds until an operation times out
RshInitialize	Initialize the library and validate the specified license key at runtime
RshIsBlocking	Determine if the client is blocked, waiting for information
RshIsConnected	Determine if the client is connected to the server
RshIsReadable	Determine if data can be read from the server
RshIsWritable	Determine if data can be written to the server
RshLogin	Establish a login session with the specified server
RshRead	Read data returned by the server
RshRegisterEvent	Register an event callback function
RshSearch	Search for a specific character sequence in the data stream
RshSetLastError	Set the last error code
RshSetTimeout	Set the number of seconds until an operation times out
RshUninitialize	Terminate use of the library by the application
RshWrite	Write data to the server

RshAsyncExecute Function

```
HCLIENT WINAPI RshAsyncExecute(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCommand,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **RshAsyncExecute** function is used to establish a connection with the server and execute the specified command.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **RshExecute** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. One of the following values should be used:

Constant	Description
RSH_PORT_REXEC	A connection is established with the server using port 512, the rexec service. This service requires that the client provide a username and password to execute the specified command.
RSH_PORT_RSHELL	A connection is established with the server using port 514, the rshell service. This service uses host equivalence to authenticate the user. With host equivalence, the server considers the client to be equivalent to itself, and as long as the specified user exists on the server, the client is permitted to execute commands on behalf of the user without requiring a password. Host equivalence is configured by the server administrator.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
RSH_OPTION_RESERVEDPORT	This option specifies that a reserved port should be used to establish the connection. Reserved ports are those port numbers which are less than 1024. This option should be specified when connecting on the RSH_PORT_RSHELL port.
RSH_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
RSH_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpszUserName

A pointer to a string which specifies the username used to authenticate the client session.

lpszPassword

A pointer to a string which specifies the password used to authenticate the client session. This parameter is only used when connecting to the RSH_PORT_REXEC port. If the password is not required, this parameter may be NULL.

lpszCommand

A pointer to a string which specifies the command to execute on the server.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur. If this parameter is NULL, then a synchronous (blocking) connection will be established with the server.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the *hEventWnd* parameter is NULL, this parameter should be specified as WM_NULL.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **RshGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the

following event identifiers may be sent:

Constant	Description
RSH_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
RSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
RSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
RSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and reconnect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To cancel asynchronous notification and return the client to a blocking mode, use the **RshDisableEvents** function.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **RshAttachThread** function.

Specifying the RSH_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RshAsyncLogin](#), [RshDisconnect](#), [RshExecute](#), [RshInitialize](#), [RshLogin](#), [RshUninitialize](#)

RshAsyncLogin Function

```
HCLIENT WINAPI RshAsyncLogin(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwReserved,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszTerminal,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **RshAsyncLogin** function is used to establish a terminal session with the server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **RshLogin** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port should be used.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwReserved

A reserved parameter. This value should always be zero.

lpszUserName

A pointer to a string which specifies the username used to authenticate the client session.

lpszTerminal

A pointer to a string which specifies the terminal type which the client will be identified as using during the session. If no particular terminal emulation is required, this parameter may be NULL.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur. If this parameter is NULL, then a synchronous (blocking) connection will be established with the server.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the *hEventWnd*

parameter is NULL, this parameter should be specified as WM_NULL.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **RshGetLastError**.

Remarks

The **RshAsyncLogin** function uses host equivalence, where the client is permitted to login without requiring a password. Host equivalence must be configured by the server administrator and it is typically restricted to specific users. Note that if configured improperly, host equivalence can introduce a significant security loophole. Refer to your UNIX system documentation for more information about host equivalence and the various remote command services.

On UNIX based systems, the terminal name specified by the *lpszTerminal* parameter corresponds to a termcap or terminfo entry as set in the TERM environment variable. On Windows based systems which implement the rlogin service, this parameter may be ignored and the server will assume that the client is capable of displaying ANSI escape sequences. On VMS systems, the terminal name should correspond to the terminal type used with the SET TERMINAL/DEVICE command.

If this parameter is passed as NULL pointer or an empty string, a default terminal type named "unknown" will be used. On most UNIX and VMS systems this defines a terminal which is not capable of cursor positioning using control or escape sequences. This terminal type may not be recognized and an error may be displayed when the user logs in indicating that the terminal type is invalid.

Refer to the documentation for the server system to determine what terminal type names are available to you. Remember that on UNIX systems, the terminal type is case-sensitive. Some of the more common terminal types are:

Terminal Type	Description
ansi	This terminal type is usually available on UNIX based servers. This specifies that the client is capable of displaying standard ANSI escape sequences for cursor control.
dumb	This terminal type typically specifies a terminal display which does not support control or escape sequences for cursor positioning. If you do not want escape sequences embedded in the data stream and the server returns an error if the terminal type is not specified, try using this terminal type.
pcansi	This terminal type is usually available on UNIX based servers. This specifies that the client is using a PC terminal emulator that supports basic ANSI escape sequences for cursor control. This may also enable escape sequences which can set the display colors.
vt100	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT100. This specifies that the client is capable of emulating a DEC VT100 terminal. The VT100 supports many of the same cursor control sequences as an ANSI terminal.
vt220	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-

	VT220. This specifies that the client is capable of emulating a DEC VT220 terminal, which is a later version of the VT100.
vt320	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT320. This specifies that the client is capable of emulating a DEC VT320 terminal, which is similar to the VT100 and VT220 and provides advanced features such as the ability to set display colors.
xterm	This terminal type is may be available on UNIX based servers which have X Windows installed. This specifies that the client is a using the X Windows xterm emulator which supports standard ANSI escape sequences for cursor control.

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
RSH_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
RSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
RSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
RSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To cancel asynchronous notification and return the client to a blocking mode, use the **RshDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RshAsyncExecute](#), [RshDisconnect](#), [RshExecute](#), [RshInitialize](#), [RshLogin](#), [RshUninitialize](#)

RshAttachThread Function

```
DWORD WINAPI RshAttachThread(  
    HCLIENT hClient  
    DWORD dwThreadId  
);
```

The **RshAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **RshAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **RshAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **RshCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **RshUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[RshCancel](#), [RshAsyncExecute](#), [RshAsyncLogin](#), [RshDisconnect](#), [RshExecute](#), [RshInitialize](#), [RshLogin](#), [RshUninitialize](#)

RshCancel Function

```
INT WINAPI RshCancel(  
    HCLIENT hClient  
);
```

The **RshCancel** function cancels any outstanding blocking operation in the client, causing the blocking function to fail. The application may then retry the operation or terminate the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero.

If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Remarks

When the **RshCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csrshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RshIsBlocking](#)

RshDisableEvents Function

```
INT WINAPI RshDisableEvents(  
    HCLIENT hClient  
);
```

The **RshDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Remarks

This function affects both event notification and event callbacks. Any outstanding events in the message queue should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[RshEnableEvents](#), [RshFreezeEvents](#), [RshRegisterEvent](#)

RshDisableTrace Function

```
BOOL WINAPI RshDisableTrace();
```

The **RshDisableTrace** function disables the logging of socket function calls to the trace log file.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csrshv10.lib

See Also

[RshEnableTrace](#)

RshDisconnect Function

```
INT WINAPI RshDisconnect(  
    HCLIENT hClient  
);
```

The **RshDisconnect** function terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[RshAsyncExecute](#), [RshAsyncLogin](#), [RshExecute](#), [RshInitialize](#), [RshLogin](#), [RshUninitialize](#)

RshEnableEvents Function

```
INT WINAPI RshEnableEvents(  
    HCLIENT hClient,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **RshEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RshRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the event notification window. This window receives a user-defined message which specifies the event that has occurred. If this value is NULL, event notification is disabled.

uEventMsg

An unsigned integer which specifies the user-defined message that is sent when an event occurs. This parameter's value must be greater than the value of WM_USER.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
RSH_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
RSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
RSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_WRITE	The client can now write data. This notification is sent after a

	connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
RSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and reconnect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To cancel asynchronous notification and return the client to a blocking mode, use the **RshDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[RshDisableEvents](#), [RshFreezeEvents](#), [RshRegisterEvent](#)

RshEnableTrace Function

```
BOOL WINAPI RshEnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **RshEnableTrace** function enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace function logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RshDisableTrace](#)

RshEventProc Function

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

The **RshEventProc** function is an application-defined callback function that processes events generated by the calling process.

Parameters

hClient

The handle to the client session.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **RshRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **RshRegisterEvent** function. The **RshEventProc** function is a placeholder for the application-defined function name.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[RshDisableEvents](#), [RshEnableEvents](#), [RshFreezeEvents](#), [RshRegisterEvent](#)

RshExecute Function

```
HCLIENT WINAPI RshExecute(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCommand  
);
```

The **RshExecute** function is used to establish a connection with the server and execute the specified command.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. One of the following values should be used:

Constant	Description
RSH_PORT_REXEC	A connection is established with the server using port 512, the rexec service. This service requires that the client provide a username and password to execute the specified command.
RSH_PORT_RSHELL	A connection is established with the server using port 514, the rshell service. This service uses host equivalence to authenticate the user. With host equivalence, the server considers the client to be equivalent to itself, and as long as the specified user exists on the server, the client is permitted to execute commands on behalf of the user without requiring a password. Host equivalence is configured by the server administrator.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
RSH_OPTION_RESERVEDPORT	This option specifies that a reserved port should be used to establish the connection. Reserved ports are those port numbers which are less than 1024. This option should be specified when connecting on the RSH_PORT_RSHELL port.
RSH_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both

	an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
RSH_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpszUserName

A pointer to a string which specifies the username used to authenticate the client session.

lpszPassword

A pointer to a string which specifies the password used to authenticate the client session. This parameter is only used when connecting to the RSH_PORT_REXEC port. If the password is not required, this parameter may be NULL.

lpszCommand

A pointer to a string which specifies the command to execute on the server.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **RshGetLastError**.

Remarks

To prevent this function from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **RshExecute** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **RshAttachThread** function.

Specifying the RSH_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RshDisconnect](#), [RshInitialize](#), [RshLogin](#), [RshSearch](#), [RshUninitialize](#)

RshFreezeEvents Function

```
INT WINAPI RshFreezeEvents(  
    HCLIENT hClient,  
    BOOL bFreeze  
);
```

The **RshFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[RshDisableEvents](#), [RshEnableEvents](#), [RshRegisterEvent](#)

RshGetErrorString Function

```
INT WINAPI RshGetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);
```

The **RshGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **RshGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The last-error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csrshv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RshGetLastError](#), [RshSetLastError](#)

RshGetLastError Function

```
DWORD WINAPI RshGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **RshSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **RshGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **RshSetLastError(0)** when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or RSH_ERROR. Those functions which call **RshSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[RshGetErrorString](#), [RshSetLastError](#)

RshGetStatus Function

```
INT WINAPI RshGetStatus(  
    HCLIENT hClient  
);
```

The **RshGetStatus** function returns the current status of the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Remarks

The **RshGetStatus** function returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Constant	Description
1	RSH_STATUS_IDLE	The client is current idle and not sending or receiving data.
2	RSH_STATUS_CONNECT	The client is establishing a connection with the server.
3	RSH_STATUS_READ	The client is reading data from the server.
4	RSH_STATUS_WRITE	The client is writing data to the server.
5	RSH_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[RshIsBlocking](#), [RshIsConnected](#), [RshIsReadable](#)

RshGetTimeout Function

```
INT WINAPI RshGetTimeout(  
    HCLIENT hClient  
);
```

The **RshGetTimeout** function returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[RshSetTimeout](#)

RshInitialize Function

```
BOOL WINAPI RshInitialize(  
    LPCTSTR lpszLicenseKey,  
    LPINITDATA lpData  
);
```

The **RshInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpData

Pointer to an INITDATA data structure. This parameter may be NULL if the initialization data for the library is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **RshGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must insure that this function is called with a valid license key.

If the *lpData* argument is specified, it must point to an INITDATA structure which has been initialized by setting the *dwSize* member to the size of the structure. All other structure members should be set to zero. If the function is successful, then the INITDATA structure will be filled with identifying information about the library.

Although it is only required that **RshInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **RshUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **RshInitialize** or **RshUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

RshIsBlocking Function

```
BOOL WINAPI RshIsBlocking(  
    HCLIENT hClient  
);
```

The **RshIsBlocking** function is used to determine if the client is currently performing a blocking operation.

Parameters

hClient

Handle to the client session.

Return Value

If the client is performing a blocking operation, the function returns TRUE. If the client is not performing a blocking operation, or the client handle is invalid, the function returns FALSE.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking function may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **RshIsBlocking** function can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[RshCancel](#), [RshGetStatus](#), [RshIsConnected](#), [RshIsReadable](#)

RshIsConnected Function

```
BOOL WINAPI RshIsConnected(  
    HCLIENT hClient  
);
```

The **RshIsConnected** function is used to determine if the client is currently connected to a server.

Parameters

hClient

Handle to the client session.

Return Value

If the client is connected to a server, the function returns a non-zero value. If the client is not connected, or the client handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[RshGetStatus](#), [RshIsBlocking](#), [RshIsReadable](#), [RshIsWritable](#)

RshIsReadable Function

```
BOOL WINAPI RshIsReadable(  
    HCLIENT hClient,  
    INT nTimeout,  
    LPDWORD lpdwAvail  
);
```

The **RshIsReadable** function is used to determine if data is available to be read from the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the timeout period, the function returns a non-zero value. If the client cannot read any data, the function returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csrshv10.lib

See Also

[RshGetStatus](#), [RshIsBlocking](#), [RshIsConnected](#), [RshIsWritable](#), [RshRead](#)

RshIsWritable Function

```
BOOL WINAPI RshIsWritable(  
    HCLIENT hClient,  
    INT nTimeout  
);
```

The **RshIsWritable** function is used to determine if data can be written to the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the function returns a non-zero value. If the client cannot write any data, the function returns zero.

Remarks

Although this function can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csrshv10.lib

See Also

[RshGetStatus](#), [RshIsBlocking](#), [RshIsConnected](#), [RshIsReadable](#), [RshWrite](#)

RshLogin Function

```
HCLIENT WINAPI RshLogin(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwReserved,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszTerminal  
);
```

The **RshLogin** function is used to establish a terminal session with the server.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port should be used.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwReserved

A reserved parameter. This value should always be zero.

lpszUserName

A pointer to a string which specifies the username used to authenticate the client session.

lpszTerminal

A pointer to a string which specifies the terminal type which the client will be identified as using during the session. If no particular terminal emulation is required, this parameter may be NULL.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is `INVALID_CLIENT`. To get extended error information, call **RshGetLastError**.

Remarks

To prevent this function from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **RshLogin** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

The **RshLogin** function uses host equivalence, where the client is permitted to login without requiring a password. Host equivalence must be configured by the server administrator and it is typically restricted to specific users. Note that if configured improperly, host equivalence can introduce a significant security loophole. Refer to your UNIX system documentation for more information about host equivalence and the various remote command services. Note that if there is no host equivalence for the local host and the user account requires a password, the system will prompt for a password.

On UNIX based systems, the terminal name specified by the *lpszTerminal* parameter corresponds

to a termcap or terminfo entry as set in the TERM environment variable. On Windows based systems which implement the rlogin service, this parameter may be ignored and the server will assume that the client is capable of displaying ANSI escape sequences. On VMS systems, the terminal name should correspond to the terminal type used with the SET TERMINAL/DEVICE command.

If this parameter is passed as NULL pointer or an empty string, a default terminal type named "unknown" will be used. On most UNIX and VMS systems this defines a terminal which is not capable of cursor positioning using control or escape sequences. This terminal type may not be recognized and an error may be displayed when the user logs in indicating that the terminal type is invalid.

Refer to the documentation for the server system to determine what terminal type names are available to you. Remember that on UNIX systems, the terminal type is case-sensitive. Some of the more common terminal types are:

Terminal Type	Description
ansi	This terminal type is usually available on UNIX based servers. This specifies that the client is capable of displaying standard ANSI escape sequences for cursor control.
dumb	This terminal type typically specifies a terminal display which does not support control or escape sequences for cursor positioning. If you do not want escape sequences embedded in the data stream and the server returns an error if the terminal type is not specified, try using this terminal type.
pcansi	This terminal type is usually available on UNIX based servers. This specifies that the client is using a PC terminal emulator that supports basic ANSI escape sequences for cursor control. This may also enable escape sequences which can set the display colors.
vt100	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT100. This specifies that the client is capable of emulating a DEC VT100 terminal. The VT100 supports many of the same cursor control sequences as an ANSI terminal.
vt220	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT220. This specifies that the client is capable of emulating a DEC VT220 terminal, which is a later version of the VT100.
vt320	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT320. This specifies that the client is capable of emulating a DEC VT320 terminal, which is similar to the VT100 and VT220 and provides advanced features such as the ability to set display colors.
xterm	This terminal type is may be available on UNIX based servers which have X Windows installed. This specifies that the client is using the X Windows xterm emulator which supports standard ANSI escape sequences for cursor control.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RshDisconnect](#), [RshExecute](#), [RshInitialize](#), [RshSearch](#), [RshUninitialize](#)

RshRead Function

```
INT WINAPI RshRead(  
    HCLIENT hClient,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **RshRead** function reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hClient

Handle to the client session.

lpBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Remarks

When **RshRead** is called and the client is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csrshv10.lib

See Also

[RshIsBlocking](#), [RshIsReadable](#), [RshSearch](#), [RshWrite](#)

RshRegisterEvent Function

```
INT WINAPI RshRegisterEvent(  
    HCLIENT hClient,  
    UINT nEventId,  
    INETEVENTPROC LpfnEvent,  
    DWORD_PTR dwParam  
);
```

The **RshRegisterEvent** function registers an event handler for the specified event.

Parameters

hClient

Handle to client session.

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One or more of the following values may be used:

Constant	Description
RSH_EVENT_CONNECT	The connection to the server has completed.
RSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
RSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
RSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
RSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

lpfnEvent

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **RshEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the

x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Remarks

The **RshRegisterEvent** function associates a callback function with a specific event. The event handler is an **RshEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[RshDisableEvents](#), [RshEnableEvents](#), [RshEventProc](#), [RshFreezeEvents](#)

RshSearch Function

```
BOOL WINAPI RshSearch(  
    HCLIENT hClient,  
    LPCTSTR lpszString,  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwReserved  
);
```

The **RshSearch** function searches for a specific character sequence in the data stream and stops reading if the sequence is encountered.

Parameters

hClient

Handle to the client session.

lpszString

A pointer to a string which specifies the sequence of characters to search for in the data stream. This parameter cannot be NULL or point to an empty string.

lpvBuffer

A pointer to a byte buffer which will contain the output from the server, or a pointer to a global memory handle which will reference the output when the function returns. If the output from the server is not required, this parameter may be NULL.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvBuffer* parameter. If the *lpvBuffer* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of output stored in the buffer. If the *lpvBuffer* parameter is NULL, this parameter should also be NULL.

dwReserved

A reserved parameter. This value must be zero.

Return Value

If the function succeeds and the character sequences was found in the data stream, the return value is non-zero. If the function fails or a timeout occurs before the sequence is found, the return value is zero. To get extended error information, call **RshGetLastError**.

Remarks

The **RshSearch** function searches for a character sequence in the data stream and stops reading when it is found. This is useful when the client wants to automate responses to the server, such as executing a command and processing the output. The function collects the output from the server and stores it in the buffer specified by the *lpvBuffer* parameter. When the function returns, the buffer will contain everything sent by the server up to and including the search string.

The *lpvBuffer* parameter may be specified in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the a fixed amount of output. In this case, the *lpvBuffer* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer. If the server sends more output than can be stored in the buffer, the remaining output will be discarded.

The second method that can be used is have the *lpvBuffer* parameter point to a global memory handle which will contain the output when the function returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. This method is preferred if the client application does not have a general idea of how much output will be generated until the search string is found.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[RshExecute](#), [RshIsBlocking](#), [RshIsReadable](#), [RshLogin](#), [RshRead](#)

RshSetLastError Function

```
VOID WINAPI RshSetLastError(  
    DWORD dwErrorCode  
);
```

The **RshSetLastError** function sets the error code for the current thread. This function is typically used to clear the last error by passing a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or RSH_ERROR. Those functions which call **RshSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **RshGetLastError** function. The use of **RshGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csrshv10.lib

See Also

[RshGetErrorString](#), [RshGetLastError](#)

RshSetTimeout Function

```
INT WINAPI RshSetTimeout(  
    HCLIENT hClient,  
    INT nTimeout  
);
```

The **RshSetTimeout** function sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[RshGetTimeout](#)

RshUninitialize Function

```
VOID WINAPI RshUninitialize();
```

The **RshUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **RshInitialize** call before it can call any of the other the library functions. When it has completed the use of library, the application must call **RshUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There must be a call to **RshUninitialize** for every successful call to **RshInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csrshv10.lib

See Also

[RshDisconnect](#), [RshInitialize](#)

RshWrite Function

```
INT WINAPI RshWrite(  
    HCLIENT hClient,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **RshWrite** function sends the specified number of bytes to the server.

Parameters

hClient

Handle to the client session.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is RSH_ERROR. To get extended error information, call **RshGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the RSH_EVENT_WRITE asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csrshv10.lib

See Also

[RshIsBlocking](#), [RshIsWritable](#), [RshRead](#)

Remote Command Protocol Data Structures

- INITDATA

INITDATA Structure

This structure contains information about the SocketTools library when the initialization function returns.

```
typedef struct _INITDATA
{
    DWORD      dwSize;
    DWORD      dwVersionMajor;
    DWORD      dwVersionMinor;
    DWORD      dwVersionBuild;
    DWORD      dwOptions;
    DWORD_PTR  dwReserved1;
    DWORD_PTR  dwReserved2;
    TCHAR      szDescription[128];
} INITDATA, *LPINITDATA;
```

Members

dwSize

Size of this structure. This structure member must be set to the size of the structure prior to calling the initialization function. Failure to do so will cause the function to fail.

dwVersionMajor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the major version number for the library.

dwVersionMinor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the minor version number for the library.

dwVersionBuild

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the build number for the library.

dwOptions

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved1

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved2

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

szDescription

A null terminated string which describes the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

Secure Shell Protocol Library

Establish an interactive terminal session with an SSH server and execute remote commands.

Reference

- [Functions](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

File Name	CSTSHV10.DLL
Version	10.0.1468.2518
LibID	4C76F231-8ED9-4850-B882-01B77485F43A
Import Library	CSTSHV10.LIB
Dependencies	None
Standards	RFC 4251

Remarks

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

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it

should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

Secure Shell Protocol Functions

Function	Description
SshAsyncConnect	Connect asynchronously to the specified server
SshAttachThread	Attach the specified client handle to another thread
SshCancel	Cancel the current blocking operation
SshConnect	Connect to the specified server
SshControl	Send a control sequence to the server
SshCreateSecurityCredentials	Create a new security credentials structure
SshDeleteSecurityCredentials	Delete a previously created security credentials structure
SshDisableEvents	Disable asynchronous event notification
SshDisableTrace	Disable logging of socket function calls to the trace log
SshDisconnect	Disconnect from the current server
SshEnableEvents	Enable asynchronous event notification
SshEnableTrace	Enable logging of socket function calls to a file
SshEventProc	Callback function that processes events generated by the client
SshExecute	Execute a command on a server and return the output in the specified buffer
SshFreezeEvents	Suspend asynchronous event processing
SshGetErrorString	Return a description for the specified error code
SshGetExitCode	Return the exit code from the remote program
SshGetLastError	Return the last error code
SshGetLineMode	Return the current mode used to send end-of-line character sequences
SshGetSecurityInformation	Return security information about the current client connection
SshGetStatus	Return the current client status
SshGetTimeout	Return the number of seconds until an operation times out
SshInitialize	Initialize the library and validate the specified license key at runtime
SshIsBlocking	Determine if the client is blocked, waiting for information
SshIsConnected	Determine if the client is connected to the server
SshIsReadable	Determine if data can be read from the server
SshIsWritable	Determine if data can be written to the server
SshPeek	Read data returned by the server, but do not remove it from the receive buffer
SshRead	Read data returned by the server
SshReadLine	Read a line of text from the server and return it in a string buffer

SshRegisterEvent	Register an event callback function
SshSearch	Search for a specific character sequence in the data stream
SshSetLastError	Set the last error code
SshSetLineMode	Change how end-of-line character sequences are sent to the server
SshSetTimeout	Set the number of seconds until an operation times out
SshUninitialize	Terminate use of the library by the application
SshWrite	Write data to the server
SshWriteLine	Write a line of text to the server

SshAsyncConnect Function

```
HCLIENT WINAPI SshAsyncConnect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSSHOPTIONDATA lpOptions,  
    LPSECURITYCREDENTIALS lpCredentials,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **SshAsyncConnect** function is used to establish a connection with the server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

lpszRemoteHost

A pointer to a string which specifies the name of the server to connect to. This may either be a fully-qualified domain name or an IP address. This parameter cannot be NULL.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number 22 should be used.

lpszUserName

A pointer to a string which specifies the user name which will be used to authenticate the client session. This parameter must specify a valid user name and cannot be NULL or an empty string.

lpszPassword

A pointer to a string which specifies the password which will be used to authenticate the client session. If the user does not have a password, this parameter can be NULL or an empty string.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SSH_OPTION_NONE	No options specified. A standard terminal session will be established with the default terminal type.
SSH_OPTION_KEEPALIVE	This option specifies the library should attempt to maintain an idle client session for long periods of time. This option is only necessary if you expect that the

	connection will be held open for more than two hours.
SSH_OPTION_NOPTY	This option specifies that a pseudoterminal (PTY) should not be created for the client session. This option is automatically set if the SSH_OPTION_COMMAND option has been specified.
SSH_OPTION_NOSHELL	This option specifies that a command shell should not be used when executing a command on the server.
SSH_OPTION_NOAUTHRSA	This option specifies that RSA authentication should not be used with SSH-1 connections. This option is ignored with SSH-2 connections and should only be specified if required by the server.
SSH_OPTION_NOPWDNULL	This option specifies the user password cannot be terminated with a null character. This option is ignored with SSH-2 connections and should only be specified if required by the server.
SSH_OPTION_NOREKEY	This option specifies the client should never attempt a repeat key exchange with the server. Some SSH servers do not support rekeying the session, and this can cause the client to become non-responsive or abort the connection after being connected for an hour.
SSH_OPTION_COMPATSID	This compatibility option changes how the session ID is handled during public key authentication with older SSH servers. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions.
SSH_OPTION_COMPATHMAC	This compatibility option changes how the HMAC authentication codes are generated. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions.
SSH_OPTION_TERMINAL	This option specifies the client session will use terminal emulation and the SSHOPTIONDATA structure specifies the characteristics of the virtual terminal. This enables the caller to specify the dimensions of the virtual display (in columns and rows) and the type of terminal that will be emulated. If this option is omitted, the session will default to a virtual display that is 80 columns, 25 rows.
SSH_OPTION_COMMAND	This option specifies the client session will be used to issue a command that is executed on the server, and the output will be returned to the caller. If this option is specified, the session will not be interactive and no pseudoterminal is created for the client. The szCommandLine member of the SSHOPTIONDATA structure specifies the command string that will be sent to the server.

SSH_OPTION_PROXYSERVER	This option specifies the client should establish a connection through a proxy server. The two protocols that are supported are SSH_PROXY_HTTP and SSH_PROXY_TELNET, which specifies the protocol that the proxy connection is created through. The proxy-related members of the SSHOPTIONDATA structure should be set to the appropriate values.
SSH_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
SSH_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpOptions

A pointer to a [SSHOPTIONDATA](#) structure which specifies additional information for one or more options. If no optional data is required, a NULL pointer may be specified.

lpCredentials

A pointer to a [SECURITYCREDENTIALS](#) structure which specifies additional security-related information required to establish the connection. This parameter may be NULL, in which case default values will be used. Note that the *dwSize* member must be initialized to the size of the **SECURITYCREDENTIALS** structure that is being passed to the function.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **SshGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description

SSH_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
SSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
SSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
SSH_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
SSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
SSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and reconnect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To cancel asynchronous notification and return the client to a blocking mode, use the **SshDisableEvents** function.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create worker threads for each additional session if more than one active client session is required.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **SshAttachThread** function.

Specifying the SSH_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Example

```
// Define the asynchronous event notification message ID
#define WM_CLIENT_EVENTS (WM_APP + 1)

HCLIENT hClient;
SSHOPTIONDATA sshOptions;

// Initialize the SSHOPTIONDATA structure and specify the
```

```

// command that should be executed on the server
ZeroMemory(&sshOptions, sizeof(sshOptions));
lstrcpyn(sshOptions.szCommandLine, lpszCommand, SSH_MAXCOMMANDLEN);

// Establish a connection with the SSH server

hClient = SshAsyncConnect(lpszHostName,
                        SSH_PORT_DEFAULT,
                        lpszUserName,
                        lpszPassword,
                        SSH_TIMEOUT,
                        SSH_OPTION_COMMAND,
                        &sshOptions,
                        NULL,
                        hAppWnd,
                        WM_CLIENT_EVENTS);

// If the connection attempt fails, then get a description of
// the error and display it in a message box

if (hClient == INVALID_CLIENT)
{
    DWORD dwError;
    TCHAR szError[128];

    dwError = SshGetLastError();
    if (dwError > 0)
    {
        SshGetErrorString(dwError, szError, 128);
        MessageBox(NULL, szError, _T("Error"), MB_ICONEXCLAMATION);
    }

    return;
}

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SshConnect](#), [SshCreateSecurityCredentials](#), [SshDeleteSecurityCredentials](#), [SshDisconnect](#), [SshInitialize](#), [SshUninitialize](#), [SECURITYCREDENTIALS](#), [SSH_OPTIONDATA](#)

SshAttachThread Function

```
DWORD WINAPI SshAttachThread(  
    HCLIENT hClient  
    DWORD dwThreadId  
);
```

The **SshAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **SshAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **SshAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **SshCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **SshUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[SshCancel](#), [SshAsyncConnect](#), [SshConnect](#), [SshDisconnect](#), [SshUninitialize](#)

SshCancel Function

```
INT WINAPI SshCancel(  
    HCLIENT hClient  
);
```

The **SshCancel** function cancels any outstanding blocking operation in the client, causing the blocking function to fail. The application may then retry the operation or terminate the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

When the **SshCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[SshControl](#), [SshIsBlocking](#)

SshConnect Function

```
HCLIENT WINAPI SshConnect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSSHOPTIONDATA lpOptions,  
    LPSECURITYCREDENTIALS lpCredentials  
);
```

The **SshConnect** function is used to establish a connection with the server.

Parameters

lpszRemoteHost

A pointer to a string which specifies the name of the server to connect to. This may either be a fully-qualified domain name or an IP address. This parameter cannot be NULL.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number 22 should be used.

lpszUserName

A pointer to a string which specifies the user name which will be used to authenticate the client session. This parameter must specify a valid user name and cannot be NULL or an empty string.

lpszPassword

A pointer to a string which specifies the password which will be used to authenticate the client session. If the user does not have a password, this parameter can be NULL or an empty string.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SSH_OPTION_NONE	No options specified. A standard terminal session will be established with the default terminal type.
SSH_OPTION_KEEPAIVE	This option specifies the library should attempt to maintain an idle client session for long periods of time. This option is only necessary if you expect that the connection will be held open for more than two hours.
SSH_OPTION_NOPTY	This option specifies that a pseudoterminal (PTY) should not be created for the client session. This option is automatically set if the SSH_OPTION_COMMAND option has been specified.
SSH_OPTION_NOSHELL	This option specifies that a command shell should not

	be used when executing a command on the server.
SSH_OPTION_NOAUTHRSA	This option specifies that RSA authentication should not be used with SSH-1 connections. This option is ignored with SSH-2 connections and should only be specified if required by the server.
SSH_OPTION_NOPWDNULL	This option specifies the user password cannot be terminated with a null character. This option is ignored with SSH-2 connections and should only be specified if required by the server.
SSH_OPTION_NOREKEY	This option specifies the client should never attempt a repeat key exchange with the server. Some SSH servers do not support rekeying the session, and this can cause the client to become non-responsive or abort the connection after being connected for an hour.
SSH_OPTION_COMPATSID	This compatibility option changes how the session ID is handled during public key authentication with older SSH servers. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions.
SSH_OPTION_COMPATHMAC	This compatibility option changes how the HMAC authentication codes are generated. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions.
SSH_OPTION_TERMINAL	This option specifies the client session will use terminal emulation and the SSHOPTIONDATA structure specifies the characteristics of the virtual terminal. This enables the caller to specify the dimensions of the virtual display (in columns and rows) and the type of terminal that will be emulated. If this option is omitted, the session will default to a virtual display that is 80 columns, 25 rows.
SSH_OPTION_COMMAND	This option specifies the client session will be used to issue a command that is executed on the server, and the output will be returned to the caller. If this option is specified, the session will not be interactive and no pseudoterminal is created for the client. The szCommandLine member of the SSHOPTIONDATA structure specifies the command string that will be sent to the server.
SSH_OPTION_PROXYSERVER	This option specifies the client should establish a connection through a proxy server. The two protocols that are supported are SSH_PROXY_HTTP and SSH_PROXY_TELNET, which specifies the protocol that the proxy connection is created through. The proxy-related members of the SSHOPTIONDATA structure

	should be set to the appropriate values.
SSH_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
SSH_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpOptions

A pointer to a [SSH_OPTIONDATA](#) structure which specifies additional information for one or more options. If no optional data is required, a NULL pointer may be specified.

lpCredentials

A pointer to a [SECURITY_CREDENTIALS](#) structure which specifies additional security-related information required to establish the connection. This parameter may be NULL, in which case default values will be used. Note that the *dwSize* member must be initialized to the size of the **SECURITY_CREDENTIALS** structure that is being passed to the function.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is **INVALID_CLIENT**. To get extended error information, call **SshGetLastError**.

Remarks

To prevent this function from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **SshConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **SshAttachThread** function.

Specifying the **SSH_OPTION_FREETHREAD** option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Example

```
HCLIENT hClient;
SSH_OPTIONDATA sshOptions;
```

```

// Initialize the SSHOPTIONDATA structure and specify the
// command that should be executed on the server
ZeroMemory(&sshOptions, sizeof(sshOptions));
lstrcpyn(sshOptions.szCommandLine, lpszCommand, SSH_MAXCOMMANDLEN);

// Establish a connection with the SSH server

hClient = SshConnect(lpszHostName,
                    SSH_PORT_DEFAULT,
                    lpszUserName,
                    lpszPassword,
                    SSH_TIMEOUT,
                    SSH_OPTION_COMMAND,
                    &sshOptions,
                    NULL);

// If the connection attempt fails, then get a description of
// the error and display it in a message box

if (hClient == INVALID_CLIENT)
{
    DWORD dwError;
    TCHAR szError[128];

    dwError = SshGetLastError();
    if (dwError > 0)
    {
        SshGetErrorString(dwError, szError, 128);
        MessageBox(NULL, szError, _T("Error"), MB_ICONEXCLAMATION);
    }

    return;
}

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SshCreateSecurityCredentials](#), [SshDeleteSecurityCredentials](#), [SshDisconnect](#), [SshExecute](#), [SshInitialize](#), [SshUninitialize](#), [SECURITYCREDENTIALS](#), [SSHOPTIONDATA](#)

SshControl Function

```
INT WINAPI SshControl(  
    HCLIENT hClient  
    DWORD dwControlCode,  
    LPVOID lpvReserved,  
    DWORD dwReserved  
);
```

The **SshControl** function sends a control message to the server.

Parameters

hClient

Handle to the client session.

dwControlCode

A numeric control code which specifies the control message which should be sent to the server. This may be one of the following values:

Constant	Description
SSH_CONTROL_BREAK	Sends a control message to the server which simulates a break signal on a physical terminal. This is used by some operating systems as an instruction to enter a privileged configuration mode. Note that this is not the same as sending an interrupt character such as Ctrl+C to the server. This control code is ignored for SSH 1.0 sessions.
SSH_CONTROL_NOOP	Sends a control message to the server, but it does not perform any operation. This is typically used by clients to prevent the server from automatically closing a session that has been idle for a long period of time.
SSH_CONTROL_EOF	Sends a control message to the server indicating that the client has finished sending data. Note that this option is normally not used with interactive terminal sessions, and should only be used when required by the server.
SSH_CONTROL_PING	Sends a control message to the server which is used to test whether or not the server is responsive to the client. This is typically used by clients to attempt to detect if the connection to the server is still active.
SSH_CONTROL_REKEY	Sends a control message to the server requesting that the key exchange be performed again. This control code is ignored for SSH 1.0 sessions.

lpvReserved

A reserved parameter which should always be specified as NULL.

dwReserved

A reserved parameter which should always be specified as a value of 0.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is

SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

The **SshControl** function enables an application to send control messages to the server, which can cause it to take specific actions such as simulate a terminal break or request that the key exchanged be performed again. Some control messages are not supported by the SSH 1.0 protocol, in which case the control message is ignored.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[SshCancel](#), [SshIsBlocking](#)

SshCreateSecurityCredentials Function

```
BOOL WINAPI SshCreateSecurityCredentials(  
    DWORD dwProtocol,  
    DWORD dwOptions,  
    LPCTSTR lpszKeyFile,  
    LPCTSTR lpszPassword,  
    LPVOID lpvReserved,  
    LPSECURITYCREDENTIALS* lppCredentials  
);
```

The **SshCreateSecurityCredentials** function creates a **SECURITYCREDENTIALS** structure.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_SSH	Select either version 1.0 or 2.0 of the Secure Shell protocol. The actual protocol version that is selected is determined automatically. This is the recommended value.
SECURITY_PROTOCOL_SSH1	Version 1.0 of the Secure Shell protocol. This protocol has been deprecated and its use is not recommended.
SECURITY_PROTOCOL_SSH2	Version 2.0 of the Secure Shell protocol. This is currently the most commonly used version of the protocol, and most servers will require this version when establishing a connection.

dwOptions

Credentials options. This argument is reserved for future use. Set it to a value of zero when using this function.

lpszKeyFile

A pointer to a string which specifies the name of a private key file that used when authenticating the client connection. If a private key is not required, value of NULL should be specified.

lpszPassword

A pointer to a string which specifies the password for the private key file. A value of NULL specifies that no password is required.

lppCredentials

Pointer to an [LPSECURITYCREDENTIALS](#) pointer. The memory for the credentials structure will be allocated by this function and must be released by calling the **SshDeleteSecurityCredentials** function when it is no longer needed. The pointer value must be set to NULL before the function is called.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **SshGetLastError**.

Remarks

The structure that is created by this function may be used as client credentials when establishing a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **SshAsyncConnect** or **SshConnect** functions.

Example

```
LPSECURITYCREDENTIALS lpSecCred = NULL;
SshCreateSecurityCredentials(SEcurity_PROTOCOL_SSH2,
                            0,
                            lpzKeyFile,
                            lpzPassword,
                            NULL,
                            &lpSecCred);

hClient = SshConnect(lpzHostName,
                    SSH_PORT_DEFAULT,
                    lpzUserName,
                    lpzPassword,
                    SSH_TIMEOUT,
                    SSH_OPTION_DEFAULT,
                    NULL,
                    lpSecCred);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SshAsyncConnect](#), [SshConnect](#), [SshDeleteSecurityCredentials](#), [SshGetSecurityInformation](#), [SECURITYCREDENTIALS](#)

SshDeleteSecurityCredentials Function

```
VOID WINAPI SshDeleteSecurityCredentials(  
    LPSECURITYCREDENTIALS* lppCredentials  
);
```

The **SshDeleteSecurityCredentials** function deletes an existing **SECURITYCREDENTIALS** structure.

Parameters

lppCredentials

Pointer to an **LPSECURITYCREDENTIALS** pointer. On exit from the function, the pointer will be NULL.

Return Value

None.

Example

```
if (lpSecCred)  
    SshDeleteSecurityCredentials(&lpSecCred);  
  
SshUninitialize();
```

Remarks

This function can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SshCreateSecurityCredentials](#), [SshUninitialize](#)

SshDisableEvents Function

```
INT WINAPI SshDisableEvents(  
    HCLIENT hClient  
);
```

The **SshDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **SshRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

This function affects both event notification and event callbacks. Any outstanding events in the message queue should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[SshEnableEvents](#), [SshFreezeEvents](#), [SshRegisterEvent](#)

SshDisableTrace Function

```
BOOL WINAPI SshDisableTrace();
```

The **SshDisableTrace** function disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[SshEnableTrace](#)

SshDisconnect Function

```
INT WINAPI SshDisconnect(  
    HCLIENT hClient  
);
```

The **SshDisconnect** function terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SshAsyncConnect](#), [SshConnect](#), [SshUninitialize](#)

SshEnableEvents Function

```
INT WINAPI SshEnableEvents(  
    HCLIENT hClient,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **SshEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the event notification window. This window receives a user-defined message which specifies the event that has occurred. If this value is NULL, event notification is disabled.

uEventMsg

An unsigned integer which specifies the user-defined message that is sent when an event occurs. This parameter's value must be greater than the value of WM_USER. If the *hEventWnd* parameter is NULL, this value must be specified as WM_NULL.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
SSH_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
SSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
SSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
SSH_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking

	operation. This event is only generated if the client is in asynchronous mode.
SSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
SSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and reconnect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

As noted, some events are only generated when the client is asynchronous mode. These events depend on the Windows Sockets asynchronous notification mechanism.

If event notification is disabled by specifying a NULL window handle, there may still be outstanding events in the message queue that must be processed. Since event handling has been disabled, these events should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[SshDisableEvents](#), [SshFreezeEvents](#), [SshRegisterEvent](#)

SshEnableTrace Function

```
BOOL WINAPI SshEnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **SshEnableTrace** function enables the logging of Windows Sockets function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace function logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SshDisableTrace](#)

SshEventProc Function

```
VOID CALLBACK SshEventProc(  
    HCLIENT hClient,  
    UINT nEvent,  
    DWORD dwError,  
    DWORD_PTR dwParam  
);
```

The **SshEventProc** function is an application-defined callback function that processes events generated by the client.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **SshRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **SshRegisterEvent** function. The **SshEventProc** function is a placeholder for the application-defined function name.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SshDisableEvents](#), [SshEnableEvents](#), [SshFreezeEvents](#), [SshRegisterEvent](#)

SshExecute Function

```
INT WINAPI SshExecute(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCommandLine,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    LPSECURITYCREDENTIALS lpCredentials  
);
```

The **SshExecute** function executes a command on the server and returns the output in the specified buffer.

Parameters

lpszRemoteHost

A pointer to a string which specifies the name of the server. This may either be a fully-qualified domain name, or an IP address. This parameter cannot be NULL.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number 22 should be used.

lpszUserName

A pointer to a string which specifies the user name which will be used to authenticate the client session.

lpszPassword

A pointer to a string which specifies the password which will be used to authenticate the client session.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SSH_OPTION_NONE	No options specified. A standard terminal session will be established with the default terminal type.
SSH_OPTION_KEEPAIVE	This option specifies the library should attempt to maintain an idle client session for long periods of time. This option is only necessary if you expect that the connection will be held open for more than two hours.
SSH_OPTION_NOPTY	This option specifies that a pseudoterminal (PTY) should not be created for the client session. This

	option is automatically set if the SSH_OPTION_COMMAND option has been specified.
SSH_OPTION_NOSHELL	This option specifies that a command shell should not be used when executing a command on the server.
SSH_OPTION_NOAUTHRSA	This option specifies that RSA authentication should not be used with SSH-1 connections. This option is ignored with SSH-2 connections and should only be specified if required by the server.
SSH_OPTION_NOPWDNULL	This option specifies the user password cannot be terminated with a null character. This option is ignored with SSH-2 connections and should only be specified if required by the server.
SSH_OPTION_NOREKEY	This option specifies the client should never attempt a repeat key exchange with the server. Some SSH servers do not support rekeying the session, and this can cause the client to become non-responsive or abort the connection after being connected for an hour.
SSH_OPTION_COMPATSID	This compatibility option changes how the session ID is handled during public key authentication with older SSH servers. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions.
SSH_OPTION_COMPATHMAC	This compatibility option changes how the HMAC authentication codes are generated. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions.

lpvBuffer

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvBuffer* parameter. If the *lpvBuffer* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the file that was downloaded.

lpCredentials

A pointer to a [SECURITYCREDENTIALS](#) structure which specifies additional security-related information required to establish the connection. This parameter may be NULL, in which case default values will be used. Note that the *dwSize* member must be initialized to the size of the **SECURITYCREDENTIALS** structure that is being passed to the function.

Return Value

If the function succeeds, the return value is the exit code from the program that was executed on the server. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

The **SshExecute** function is used to execute a command on a server, read the output from that command and copy it into a local buffer. This function cannot be used if the connection to the server must be established through a proxy server; if a proxy server must be used, then you should use the **SshConnect** function to establish the connection, and then use either the **SshRead** or **SshReadLine** functions to read the output.

This function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the command output. In this case, the *lpvBuffer* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvBuffer* parameter point to a global memory handle which will contain the output when the function returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

When the command output is being read from the server, this function will automatically convert the data to match the end-of-line convention used on the Windows platform. This is useful when executing a command on a UNIX based system where the end-of-line is indicated by a single linefeed, while on Windows it is a carriage-return and linefeed pair. If the output contains embedded nulls or escape sequences, then this conversion will not be performed.

This function will cause the current thread to block until the command completes or a timeout occurs.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
LPBYTE lpBuffer = (LPBYTE)NULL;
DWORD cbBuffer = 0;

// Execute a command on the server and return the data into block
// of global memory allocated by the GlobalAlloc function; the handle
// to this memory will be returned in the hgblBuffer parameter
nResult = SshExecute(lpszHostName,
                    SSH_PORT_DEFAULT,
                    lpszUserName,
                    lpszPassword,
                    lpszCommandLine,
                    SSH_TIMEOUT,
                    SSH_OPTION_NONE,
                    &hgblBuffer,
                    &cbBuffer,
                    NULL);

if (nResult != SSH_ERROR)
{
    // Lock the global memory handle, returning a pointer to the
    // resource data
    lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

```
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SshConnect](#), [SshGetExitCode](#), [SshRead](#), [SshReadLine](#), [SshWrite](#), [SshWriteLine](#),
[SECURITYCREDENTIALS](#), [SSH_OPTIONDATA](#)

SshFreezeEvents Function

```
INT WINAPI SshFreezeEvents(  
    HCLIENT hClient,  
    BOOL bFreeze  
);
```

The **SshFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[SshDisableEvents](#), [SshEnableEvents](#), [SshRegisterEvent](#)

SshGetErrorString Function

```
INT WINAPI SshGetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);
```

The **SshGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **SshGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstshv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SshGetLastError](#), [SshSetLastError](#)

SshGetExitCode Function

```
INT WINAPI SshGetExitCode(  
    HCLIENT hClient  
);
```

The **SshGetExitCode** function returns the exit code for the remote session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the numeric exit code. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

This function should only be called after the command has completed and the **SshRead** function has returned a value of zero. In most cases, an exit code value of zero indicates success, while any other value indicates an error condition.

Note that the actual value is application dependent and is only meaningful in the context of that particular program. A program may choose or use exit codes in a non-standard way, such as having certain non-zero values indicate success.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[SshGetStatus](#)

SshGetLastError Function

DWORD WINAPI SshGetLastError();

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SshSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **SshGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **SshSetLastError(0)** when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or SSH_ERROR. Those functions which call **SshSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[SshGetErrorString](#), [SshSetLastError](#)

SshGetLineMode Function

```
INT WINAPI SshGetLineMode(  
    HCLIENT hClient  
);
```

The **SshGetLineMode** function returns the current line mode.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the current line mode. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

The **SshGetLineMode** function returns an integer value that specifies how end-of-line character sequences are sent to the server. For more information about how newlines are processed by the library and the available options, refer to the **SshSetLineMode** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[SshSetLineMode](#), [SshWrite](#), [SshWriteLine](#)

SshGetSecurityInformation Function

```
BOOL WINAPI SshGetSecurityInformation(  
    HCLIENT hClient,  
    LPSECURITYINFO lpSecurityInfo  
);
```

The **SshGetSecurityInformation** function returns security protocol, encryption and certificate information about the current client connection.

Parameters

hClient

Handle to the client session.

lpSecurityInfo

A pointer to a [SECURITYINFO](#) structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **SshGetLastError**.

Remarks

This function is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

The following example demonstrates how to obtain the fingerprint for the server:

```
SECURITYINFO securityInfo;  
  
securityInfo.dwSize = sizeof(SECURITYINFO);  
if (SshGetSecurityInformation(hClient, &securityInfo))  
{  
    if (securityInfo.lpszFingerprint != NULL)  
    {  
        TCHAR szMessage[256];  
        wsprintf(szMessage, _T("The fingerprint is %s",  
securityInfo.lpszFingerprint);  
        MessageBox(NULL, szMessage, "Connection", MB_OK);  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SshAsyncConnect](#), [SshConnect](#), [SshDisconnect](#), [SECURITYINFO](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

SshGetStatus Function

```
INT WINAPI SshGetStatus(  
    HCLIENT hClient  
);
```

The **SshGetStatus** function returns the current status of the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

The **SshGetStatus** function returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Constant	Description
1	SSH_STATUS_IDLE	The client is current idle and not sending or receiving data.
2	SSH_STATUS_CONNECT	The client is establishing a connection with the server.
3	SSH_STATUS_AUTHENTICATE	The client is authenticating the session with the server.
4	SSH_STATUS_READ	The client is reading data from the server.
5	SSH_STATUS_WRITE	The client is writing data to the server.
6	SSH_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstshv10.lib

See Also

[SshGetExitCode](#), [SshIsBlocking](#), [SshIsConnected](#), [SshIsReadable](#), [SshIsWritable](#)

SshGetTimeout Function

```
INT WINAPI SshGetTimeout(  
    HCLIENT hClient  
);
```

The **SshGetTimeout** function returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[SshSetTimeout](#)

SshInitialize Function

```
BOOL WINAPI SshInitialize(  
    LPCTSTR lpszLicenseKey,  
    LPINITDATA lpData  
);
```

The **SshInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpData

Pointer to an INITDATA data structure. This parameter may be NULL if the initialization data for the library is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **SshGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must insure that this function is called with a valid license key.

If the *lpData* argument is specified, it must point to an INITDATA structure which has been initialized by setting the *dwSize* member to the size of the structure. All other structure members should be set to zero. If the function is successful, then the INITDATA structure will be filled with identifying information about the library.

Although it is only required that **SshInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **SshUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **SshInitialize** or **SshUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

SshIsBlocking Function

```
BOOL WINAPI SshIsBlocking(  
    HCLIENT hClient  
);
```

The **SshIsBlocking** function is used to determine if the client is currently performing a blocking operation.

Parameters

hClient

Handle to the client session.

Return Value

If the client is performing a blocking operation, the function returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the function returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking function may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **SshIsBlocking** function can be used to determine if the client is already blocked, and if so, take some other action such as warning the user that they must wait for the operation to complete.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[SshCancel](#)

SshIsConnected Function

```
BOOL WINAPI SshIsConnected(  
    HCLIENT hClient  
);
```

The **SshIsConnected** function is used to determine if the client is currently connected to a server.

Parameters

hClient

Handle to the client session.

Return Value

If the client is connected to a server, the function returns a non-zero value. If the client is not connected, or the client handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SshIsBlocking](#), [SshIsReadable](#), [SshIsWritable](#)

SshIsReadable Function

```
BOOL WINAPI SshIsReadable(  
    HCLIENT hClient,  
    INT nTimeout,  
    LPDWORD lpdwAvail  
);
```

The **SshIsReadable** function is used to determine if data is available to be read from the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the function returns a non-zero value. If the client cannot read any data, the function returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations on different versions of Windows, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[SshGetStatus](#), [SshIsBlocking](#), [SshIsConnected](#), [SshIsWritable](#), [SshRead](#)

SshIsWritable Function

```
BOOL WINAPI SshIsWritable(  
    HCLIENT hClient,  
    INT nTimeout  
);
```

The **SshIsWritable** function is used to determine if data can be written to the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the function returns a non-zero value. If the client cannot write any data, the function returns zero.

Remarks

Although this function can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstshv10.lib

See Also

[SshGetStatus](#), [SshIsBlocking](#), [SshIsConnected](#), [SshIsReadable](#), [SshWrite](#)

SshPeek Function

```
INT WINAPI SshPeek(  
    HCLIENT hClient,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **SshPeek** function reads the specified number of bytes from the server and copies them into the buffer, but it does not remove the data from the internal receive buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hClient

Handle to the client session.

lpBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that there is no data available to be read at that time. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

The **SshPeek** function can be used to examine the data that is available to be read from the internal receive buffer. If there is no data in the receive buffer at that time, a value of zero is returned. It should be noted that this differs from the **SshRead** function, where a return value of zero indicates that there is no more data available to be read and the connection has been closed. The **SshPeek** function will never cause the client to block, and so may be safely used with asynchronous connections.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SshIsReadable](#), [SshRead](#), [SshSearch](#), [SshWrite](#)

SshRead Function

```
INT WINAPI SshRead(  
    HCLIENT hClient,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **SshRead** function reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hClient

Handle to the client session.

lpBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

When **SshRead** is called and the client is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SshIsReadable](#), [SshPeek](#), [SshReadLine](#), [SshSearch](#), [SshWrite](#), [SshWriteLine](#)

SshReadLine Function

```
BOOL WINAPI SshReadLine(  
    HCLIENT hClient,  
    LPTSTR lpszBuffer,  
    LPINT lpnLength  
);
```

The **SshReadLine** function reads up to a line of data and returns it in a string buffer.

Parameters

hSocket

Handle to the client session.

lpszBuffer

Pointer to the string buffer that will contain the data when the function returns. The string will be terminated with a null byte, and will not contain the end-of-line characters.

lpnLength

A pointer to an integer value which specifies the length of the buffer. The value should be initialized to the maximum number of characters that can be copied into the string buffer, including the terminating null character. When the function returns, its value will updated with the actual length of the string.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **SshGetLastError**.

Remarks

The **SshReadLine** function reads data sent by the server and copies it into a specified string buffer. Unlike the **SshRead** function which reads arbitrary bytes of data, this function is specifically designed to return a single line of text data in a string. When an end-of-line character sequence is encountered, the function will stop and return the data up to that point. The string buffer is guaranteed to be null-terminated and will not contain the end-of-line characters.

There are some limitations when using **SshReadLine**. The function should only be used to read text, never binary data. In particular, the function will discard nulls, linefeed and carriage return control characters. The Unicode version of this function will return a Unicode string, however this function does not support reading raw Unicode data from the server. The data is internally buffered as octets (eight-bit bytes) and converted to Unicode using the **MultiByteToWideChar** function.

This function will force the thread to block until an end-of-line character sequence is processed, the read operation times out or the server closes its end of the connection. If this function is called with asynchronous events enabled, it will automatically switch the client into a blocking mode, read the data and then restore the client to asynchronous operation. If another client operation is attempted while **SshReadLine** is blocked waiting for data from the server, an error will occur. It is recommended that this function only be used with blocking (synchronous) client connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **SshRead** and **SshReadLine** function calls can be intermixed, however be aware that **SshRead** will consume any data that has already been buffered by the **SshReadLine** function and this may

have unexpected results.

Unlike the **SshRead** function, it is possible for data to be returned in the buffer even if the return value is zero. Applications should also check the value of the *lpnLength* argument to determine if any data was copied into the buffer. For example, if a timeout occurs while the function is waiting for more data to arrive, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the function return value.

Example

```
TCHAR szBuffer[MAXBUFLEN];
INT nLength;
BOOL bResult;

do
{
    nLength = sizeof(szBuffer);
    bResult = SshReadLine(hSocket, szBuffer, &nLength);

    if (nLength > 0)
    {
        // Process the line of data returned in the string
        // buffer; the string is always null-terminated
    }
} while (bResult);

DWORD dwError = SshGetLastError();
if (dwError == ST_ERROR_CONNECTION_CLOSED)
{
    // The server has closed its side of the connection and
    // there is no more data available to be read
}
else if (dwError != 0)
{
    // An error has occurred while reading a line of data
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SshIsReadable](#), [SshRead](#), [SshWrite](#), [SshWriteLine](#)

SshRegisterEvent Function

```
INT WINAPI SshRegisterEvent(  
    HCLIENT hClient,  
    UINT nEvent,  
    SSHEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

The **SshRegisterEvent** function registers an event handler for the specified event.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Constant	Description
SSH_EVENT_CONNECT	The connection to the server has completed.
SSH_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
SSH_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
SSH_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
SSH_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
SSH_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **SshEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the

x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Remarks

The **SshRegisterEvent** function associates a callback function with a specific event. The event handler is an **SshEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `SSH_ERROR`. To get extended error information, call **SshGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstshv10.lib`

See Also

[SshDisableEvents](#), [SshEnableEvents](#), [SshEventProc](#), [SshFreezeEvents](#)

SshSearch Function

```
BOOL WINAPI SshSearch(  
    HCLIENT hClient,  
    LPCTSTR lpszString,  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwReserved  
);
```

The **SshSearch** function searches for a specific character sequence in the data stream and stops reading if the sequence is encountered.

Parameters

hClient

Handle to the client session.

lpszString

A pointer to a string which specifies the sequence of characters to search for in the data stream. This parameter cannot be NULL or point to an empty string.

lpvBuffer

A pointer to a byte buffer which will contain the output from the server, or a pointer to a global memory handle which will reference the output when the function returns. If the output from the server is not required, this parameter may be NULL.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvBuffer* parameter. If the *lpvBuffer* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of output stored in the buffer. If the *lpvBuffer* parameter is NULL, this parameter should also be NULL.

dwReserved

A reserved parameter. This value must be zero.

Return Value

If the function succeeds and the character sequences was found in the data stream, the return value is non-zero. If the function fails or a timeout occurs before the sequence is found, the return value is zero. To get extended error information, call **SshGetLastError**.

Remarks

The **SshSearch** function searches for a character sequence in the data stream and stops reading when it is found. This is useful when the client wants to automate responses to the server. The function collects the output from the server and stores it in the buffer specified by the *lpvBuffer* parameter. When the function returns, the buffer will contain everything sent by the server up to and including the matching string.

The *lpvBuffer* parameter may be specified in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the a fixed amount of output. In this case, the *lpvBuffer* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer. If the server sends more output than can be stored in the buffer, the remaining output will be discarded.

The second method that can be used is have the *lpvBuffer* parameter point to a global memory handle which will contain the output when the function returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. This method is preferred if the client application does not have a general idea of how much output will be generated until the search string is found.

Example

```
LPCTSTR lpszCommand = _T("/bin/ls -l\r\n");
LPCTSTR lpszPrompt = _T("$ ");
HGLOBAL hgblOutput = NULL;
DWORD cbOutput = 0;
BOOL bResult;

// Search for a command prompt issued by the server

bResult = SshSearch(hClient,
                   lpszPrompt,
                   NULL,
                   NULL,
                   0);

// If the shell prompt was found, issue the command
// and capture the output into the hgblBuffer global
// memory buffer; the cbBuffer variable will contain
// the actual number of bytes in the buffer when the
// function returns

if (bResult)
{
    SshWrite(hClient,
            (LPBYTE)lpszCommand,
            lstrlen(lpszCommand));

    bResult = SshSearch(hClient,
                       lpszPrompt,
                       &hgblOutput,
                       &cbOutput,
                       0);
}

// Write the contents of the output buffer to the
// standard output stream

if (bResult)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    if (lpBuffer)
        fwrite(lpBuffer, 1, cbOutput, stdout);

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

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SshIsBlocking](#), [SshIsReadable](#), [SshPeek](#), [SshRead](#), [SshReadLine](#), [SshWrite](#), [SshWriteLine](#)

SshSetLastError Function

```
VOID WINAPI SshSetLastError(  
    DWORD dwErrorCode  
);
```

The **SshSetLastError** function sets the last error code for the current thread. This function is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or SSH_ERROR. Those functions which call **SshSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **SshGetLastError** function. The use of **SshGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstshv10.lib

See Also

[SshGetErrorString](#), [SshGetLastError](#)

SshSetLineMode Function

```
INT WINAPI SshSetLineMode(  
    HCLIENT hClient,  
    INT nLineMode  
);
```

The **SshSetLineMode** function changes the current line mode for the client session.

Parameters

hClient

Handle to the client session.

nLineMode

An integer value which specifies how the newlines are sent by the library. It must be one of the following values:

Value	Constant	Description
0	SSH_NEWLINE_DEFAULT	There are no changes to how data is sent to the server. Any carriage return or linefeed characters that are sent using the SshWrite function will be sent as-is. The SshWriteLine function will terminate each line of text with a carriage return and linefeed (CRLF) sequence. This is the default line mode that is set when a new connection is established.
1	SSH_NEWLINE_CR	A carriage return is used as the end-of-line character. Any data sent using the SshWrite function that contains only a linefeed (LF) character or a carriage return and linefeed (CRLF) sequence to indicate the end-of-line will be replaced by a carriage return (CR) character. The SshWriteLine function will terminate each line of text with a single carriage return character.
2	SSH_NEWLINE_LF	A linefeed is used as the end-of-line character. Any data sent using the SshWrite function that contains only a carriage return (CR) character or a carriage return and linefeed (CRLF) sequence to indicate the end-of-line will be replaced by a linefeed (LF) character. The SshWriteLine function will terminate each line of text with a single linefeed character.
3	SSH_NEWLINE_CRLF	A carriage return and linefeed (CRLF) character sequence is used to indicate the end-of-line. Any data sent using the SshWrite function that contains only a carriage return (CR) or linefeed (LF) will be replaced by a carriage return and linefeed. The SshWriteLine function will terminate each line of text with a carriage return and linefeed sequence.

Return Value

If the function succeeds, the return value is the previous line mode for the client session. If the function fails, the return value is `SSH_ERROR`. To get extended error information, call **SshGetLastError**.

Remarks

When a connection is initially established with the server, it determines what characters are used to indicate the end-of-line and how they are displayed. On UNIX based systems, this is controlled by the settings for the pseudo-terminal that is allocated for the client session, and can be changed using the **stty** command. In most cases, the client line mode can be left at the default. However, in some cases you may need to change the line mode, particularly if you intend to send data from a Windows text file or copied from the clipboard.

Windows uses a carriage return and linefeed (CRLF) sequence to indicate the end-of-line and a UNIX based server may interpret that as multiple newlines. To prevent this, use the **SshSetLineMode** function to change the current line mode to **SSH_NEWLINE_CR** and the CRLF sequence in the text will be replaced by a single carriage return.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstshv10.lib`

See Also

[SshGetLineMode](#), [SshWrite](#), [SshWriteLine](#)

SshSetTimeout Function

```
INT WINAPI SshSetTimeout(  
    HCLIENT hClient,  
    INT nTimeout  
);
```

The **SshSetTimeout** function sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

See Also

[SshGetTimeout](#)

SshUninitialize Function

```
VOID WINAPI SshUninitialize();
```

The **SshUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **SshInitialize** call before it can call any of the other library functions. When it has completed the use of library, the application must call **SshUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There must be a call to **SshUninitialize** for every successful call to **SshInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SshDisconnect](#), [SshInitialize](#)

SshWrite Function

```
INT WINAPI SshWrite(  
    HCLIENT hClient,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **SshWrite** function sends the specified number of bytes to the server.

Parameters

hClient

Handle to the client session.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is SSH_ERROR. To get extended error information, call **SshGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the SSH_EVENT_WRITE asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SshIsWritable](#), [SshRead](#), [SshReadLine](#), [SshSetLineMode](#), [SshWriteLine](#)

SshWriteLine Function

```
BOOL WINAPI SshWriteLine(  
    HCLIENT hClient,  
    LPCTSTR lpszBuffer,  
    LPINT lpnLength  
);
```

The **SshWriteLine** function sends a line of text to the server, terminated by a carriage-return and linefeed.

Parameters

hClient

Handle to the client session.

lpszBuffer

The pointer to a string buffer which contains the data that will be sent to the server. All characters up to, but not including, the terminating null character will be written to the server. The data will always be terminated with a carriage-return and linefeed control character sequence. If this parameter points to an empty string or NULL pointer, then a only a carriage-return and linefeed are written to the server.

lpnLength

A pointer to an integer value which will contain the number of characters written to the server, including the carriage-return and linefeed sequence. If this information is not required, a NULL pointer may be specified.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **SshGetLastError**.

Remarks

The **SshWriteLine** function writes a line of text to the server and terminates the line with a carriage-return and linefeed control character sequence. Unlike the **SshWrite** function which writes arbitrary bytes of data to the server, this function is specifically designed to write a single line of text data from a string.

If the *lpszBuffer* string is terminated with a linefeed (LF) or carriage return (CR) character, it will be automatically converted to a standard CRLF end-of-line sequence. Because the string will be sent with a terminating CRLF sequence, the value returned in the *lpnLength* parameter will typically be larger than the original string length (reflecting the additional CR and LF characters), unless the string was already terminated with CRLF.

There are some limitations when using **SshWriteLine**. The function should only be used to send text, never binary data. In particular, the function will discard nulls and append linefeed and carriage return control characters to the data stream. The Unicode version of this function will accept a Unicode string, however this function does not support sending raw Unicode data to the server. Unicode strings will be automatically converted to UTF-8 encoding using the **WideCharToMultiByte** function and then written as a stream of bytes.

This function will force the thread to block until the complete line of text has been written, the write operation times out or the server aborts the connection. If this function is called with asynchronous events enabled, it will automatically switch the client into a blocking mode, send the

data and then restore the client to asynchronous operation. If another network operation is attempted while **SshWriteLine** is blocked sending data to the server, an error will occur. It is recommended that this function only be used with blocking (synchronous) connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **SshWrite** and **SshWriteLine** function calls can be safely intermixed.

Unlike the **SshWrite** function, it is possible for data to have been written to the server if the return value is zero. For example, if a timeout occurs while the function is waiting to send more data to the server, it will return zero; however, some data may have already been written prior to the error condition. If this is the case, the *lpnLength* argument will specify the number of characters actually written up to that point.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstshv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SshIsWritable](#), [SshRead](#), [SshReadLine](#), [SshSetLineMode](#), [SshWrite](#)

SSH Protocol Data Structures

- INITDATA
- SECURITYCREDENTIALS
- SECURITYINFO
- SSOPTIONDATA

INITDATA Structure

This structure contains information about the SocketTools library when the initialization function returns.

```
typedef struct _INITDATA
{
    DWORD      dwSize;
    DWORD      dwVersionMajor;
    DWORD      dwVersionMinor;
    DWORD      dwVersionBuild;
    DWORD      dwOptions;
    DWORD_PTR  dwReserved1;
    DWORD_PTR  dwReserved2;
    TCHAR      szDescription[128];
} INITDATA, *LPINITDATA;
```

Members

dwSize

Size of this structure. This structure member must be set to the size of the structure prior to calling the initialization function. Failure to do so will cause the function to fail.

dwVersionMajor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the major version number for the library.

dwVersionMinor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the minor version number for the library.

dwVersionBuild

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the build number for the library.

dwOptions

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved1

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved2

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

szDescription

A null terminated string which describes the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

SECURITYINFO Structure

This structure contains information about a secure connection that has been established with a server.

```
typedef struct _SECURITYINFO
{
    DWORD        dwSize;
    DWORD        dwProtocol;
    DWORD        dwCipher;
    DWORD        dwCipherStrength;
    DWORD        dwHash;
    DWORD        dwHashStrength;
    DWORD        dwKeyExchange;
    DWORD        dwCertStatus;
    SYSTEMTIME   stCertIssued;
    SYSTEMTIME   stCertExpires;
    LPCTSTR      lpszCertIssuer;
    LPCTSTR      lpszCertSubject;
    LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to **sizeof(SECURITYINFO)** prior to passing the address of this structure to the function. Note that if this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the protocol that was selected to establish the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_PROTOCOL_NONE	No security protocol has been selected. A secure connection has not been established with the server. The remaining member values in this structure are not valid and should be ignored.
SECURITY_PROTOCOL_SSH1	The Secure Shell 1.0 protocol has been selected. This protocol has been deprecated and is no longer widely used. It is not recommended that this protocol be used when establishing secure connections. This protocol can only be specified when connecting to an SSH server and is not supported with any other application protocol.
SECURITY_PROTOCOL_SSH2	The Secure Shell 2.0 protocol has been selected. This is the most commonly used version of the protocol. It is recommended that this version of the protocol be used unless the server explicitly requires the client to use an earlier version. This protocol can only be specified when connecting to an SSH server and is not supported with any other application protocol.

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
SECURITY_CIPHER_DES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher (also known as the Rijndael cipher) is a fixed block size cipher which use a key size of 128, 192 or 256 bits. This cipher is supported on Windows XP SP3 and later versions of the operating system.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected. One of the following values will be returned:

Constant	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected. One of the following values will be returned:

Constant	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.

SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography. This key exchange algorithm is only supported on Windows XP SP3 and later versions of the operating system.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server. This member only has meaning for connections using the SSL or TLS protocols.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertIssuer

A pointer to a string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertSubject

A pointer to a string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszFingerprint

A pointer to a string which contains a sequence of hexadecimal values that uniquely identify the server. This member is only used when a connection has been established using the Secure Shell (SSH) protocol.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

SECURITYCREDENTIALS Structure

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A value which specifies which security protocols are supported:

Constant	Description
SECURITY_PROTOCOL_SSH	Either version 1.0 or 2.0 of the Secure Shell protocol should be used when establishing the connection. The correct protocol is automatically selected based on the version of the protocol that is supported by the server.
SECURITY_PROTOCOL_SSH1	The Secure Shell 1.0 protocol should be used when establishing the connection. This is an older version of the protocol which should not be used unless explicitly required by the server. Most modern SSH server support version 2.0 of the protocol.
SECURITY_PROTOCOL_SSH2	The Secure Shell 2.0 protocol should be used when establishing the connection. This is the default version of the protocol that is supported by most SSH servers.

dwOptions

This structure member is reserved for use with SSL and TLS connections and should always be initialized to zero.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A pointer to a null terminated string which specifies the hostname that will be used when

validating a server certificate. This member should always be initialized as a NULL pointer for connections using the SSH protocol.

lpszUserName

A pointer to a null terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A pointer to a null terminated string which specifies the password needed to access the certificate. Currently this member is only used if a private key file has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A pointer to a null terminated string which specifies the name of the certificate store to open. This member should always be initialized as a NULL pointer for connections using the SSH protocol.

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. This member should always be initialized as a NULL pointer for connections using the SSH protocol.

lpszKeyFile

A pointer to a null terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used with the SSH protocol. If the member is NULL, then no private key is used.

Remarks

A client application typically only needs to create this structure if the server requires that the client provide a private key as part of the process of negotiating the secure session.

Note that the *lpszUserName* and *lpszPassword* members are values which are used to access the private key file. They are not the credentials which are used when establishing the connection with the server or authenticating the client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

SSHOPTIONDATA Structure

This structure specifies additional option information for the client session. A pointer to this structure can be passed to the [SshAsyncConnect](#) or [SshConnect](#) functions.

```
#define SSH_MAXTERMNAMELEN 32
#define SSH_MAXHOSTNAMELEN 128
#define SSH_MAXUSERNAMELEN 128
#define SSH_MAXPASSWORDLEN 128
#define SSH_MAXCOMMANDLEN 512

typedef struct _SSHOPTIONDATA
{
    DWORD dwSize;
    DWORD dwReserved;
    UINT nProxyType;
    UINT nProxyPort;
    TCHAR szProxyHost[SSH_MAXHOSTNAMELEN];
    TCHAR szProxyUser[SSH_MAXUSERNAMELEN];
    TCHAR szProxyPassword[SSH_MAXPASSWORDLEN];
    UINT nTermCols;
    UINT nTermRows;
    TCHAR szTermName[SSH_MAXTERMNAMELEN];
    TCHAR szCommandLine[SSH_MAXCOMMANDLEN];
} SSHOPTIONDATA, *LPSSHOPTIONDATA;
```

Members

dwSize

An unsigned integer value which specifies the size of the SSHOPTIONDATA structure. This member must be initialized prior to passing the structure to the **SshAsyncConnect** or **SshConnect** functions.

dwReserved

An unsigned integer value that is reserved for internal use, and should always be initialized to a value of zero.

nProxyType

An unsigned integer value that specifies the type of proxy that the client should connect through. This structure member is only used if the option SSH_OPTION_PROXYSERVER has been specified. Possible values are:

Constant	Description
SSH_PROXY_NONE	No proxy server should be used when establishing the connection.
SSH_PROXY_HTTP	The connection should be established on port 80 using HTTP. An alternate port number can be specified by setting the <i>nProxyPort</i> structure member to the desired value.
SSH_PROXY_TELNET	The connection should be established on port 23 using TELNET. An alternate port number can be specified by setting the <i>nProxyPort</i> structure member to the desired value.

nProxyPort

An unsigned integer value that specifies the port number which should be used to establish the

proxy connection. A value of zero specifies that the default port number appropriate for the selected protocol should be used. This structure member is only used if the option SSH_OPTION_PROXYSERVER has been specified.

szProxyHost

A null terminated string which specifies the host name or IP address of the proxy server. This structure member is only used if the option SSH_OPTION_PROXYSERVER has been specified.

szProxyUser

A null terminated string which specifies the user name which is used to authenticate the connection through the proxy server. This structure member is only used if the option SSH_OPTION_PROXYSERVER has been specified.

szProxyPassword

A null terminated string which specifies the password which is used to authenticate the connection through the proxy server. This structure member is only used if the option SSH_OPTION_PROXYSERVER has been specified.

nTermCols

An unsigned integer value which specifies the number of columns for the virtual terminal allocated for the client session. The default number of columns is 80. This structure member is only used if the option SSH_OPTION_TERMINAL has been specified.

nTermRows

An unsigned integer value which specifies the number of rows for the virtual terminal allocated for the client session. The default number of rows is 25. This structure member is only used if the option SSH_OPTION_TERMINAL has been specified.

szTermName

A null terminated string which specifies the name of the terminal emulation type. On UNIX based systems, this name typically corresponds to an entry in the terminal capability database (either termcap or terminfo). If the name is not specified, then the default name terminal name of "unknown" will be used. This structure member is only used if the option SSH_OPTION_TERMINAL has been specified.

szCommandLine

A null terminated string which specifies the command that should be executed on the server. The output from the command is returned to the client, and the session is terminated. This structure member is only used if the option SSH_OPTION_COMMAND has been specified.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

Simple Mail Transfer Protocol Library

Submit email messages for delivery to one or more recipients.

Reference

- [Functions](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

File Name	CSMTPV10.DLL
Version	10.0.1468.2518
LibID	D74CB488-F2C1-4F78-858A-201AEA7483FE
Import Library	CSMTPV10.LIB
Dependencies	None
Standards	RFC 821, RFC 1425, RFC 1869, RFC 2821

Overview

The Simple Mail Transfer Protocol (SMTP) enables applications to deliver email messages to one or more recipients. The library provides an API for addressing and delivering messages, and extended features such as user authentication and delivery status notification. Unlike Microsoft's Messaging API (MAPI) or Collaboration Data Objects (CDO), there is no requirement to have certain third-party email applications installed or specific types of servers installed on the local system. The library can be used to deliver mail through a wide variety of systems, from standard UNIX based mail servers to Windows systems running Microsoft Exchange.

Using this library, messages can be delivered directly to the recipient, or they can be routed through a relay server, such as an Internet service provider's mail system. The SocketTools Mail Message API can be integrated with this library in order to provide an extremely simple, yet flexible interface for composing and delivering messages.

This library supports secure connections using the standard SSL and TLS protocols. Both implicit and explicit SSL connections are supported, as well as client certificates used for authentication.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Simple Mail Transfer Protocol Functions

Function	Description
SmtpAddRecipient	Add an address to the recipient list
SmtpAppendMessage	Append contents of specified file to the current message
SmtpAsyncConnect	Establish an asynchronous connection with a server
SmtpAsyncSendMessage	Send message to the specified recipient
SmtpAsyncSubmitMessage	Compose and submit a message for delivery to the specified mail server
SmtpAttachThread	Attach the specified client handle to another thread
SmtpAuthenticate	Authenticate the client session with a user name and password
SmtpCancel	Cancel the current blocking operation
SmtpCloseMessage	Close the message being composed and submit for delivery
SmtpCommand	Send a command to the server
SmtpConnect	Establish a connection with a server
SmtpCreateMessage	Create a new message
SmtpCreateSecurityCredentials	Allocate a structure to establish client security credentials
SmtpDeleteSecurityCredentials	Delete the specified client security credentials
SmtpDisableEvents	Disable all event notification, including event callbacks
SmtpDisableTrace	Disable logging of socket function calls to the trace log
SmtpDisconnect	Disconnect from the current server
SmtpEnableEvents	Enable event handling by the library
SmtpEnableTrace	Enable logging of socket function calls to a file
SmtpEnumTasks	Return a list of asynchronous tasks
SmtpEventProc	Process events generated by the client
SmtpExpandAddress	Expand the specified address
SmtpFreezeEvents	Suspend and resume event handling by the client
SmtpGetCurrentDate	Return the current date and time
SmtpGetDeliveryOptions	Return the delivery options for the current session
SmtpGetErrorString	Return a description for the specified error code
SmtpGetExtendedOptions	Return the extended options supported by the server
SmtpGetLastError	Return the last error code
SmtpGetResultCode	Return the result code from the previous command
SmtpGetResultString	Return the result string from the previous command
SmtpGetSecurityInformation	Return security information about the current client connection
SmtpGetStatus	Return the current status of the client

SmtpGetTaskError	Return the last error code for the specified asynchronous task
SmtpGetTaskId	Return the unique task identifier associated with the specified client session
SmtpGetTimeout	Return the number of seconds until an operation times out
SmtpGetTransferStatus	Return data transfer statistics
SmtpInitialize	Initialize the library and validate the specified license key at runtime
SmtpIsBlocking	Determine if the client is blocked, waiting for information
SmtpIsConnected	Determine if the client is connected to the server
SmtpIsReadable	Determine if data can be read from the server
SmtpIsWritable	Determine if data can be written to the server
SmtpRegisterEvent	Register an event handler for the specified event
SmtpReset	Reset the client and return to a command state
SmtpSendMessage	Send message to the specified recipient
SmtpSetDeliveryOptions	Set the delivery options for the current session
SmtpSetLastError	Set the last error code
SmtpSetTimeout	Set the number of seconds until an operation times out
SmtpSubmitMessage	Compose and submit a message for delivery to the specified mail server
SmtpSubmitMessageEx	Compose and submit a message for delivery with additional options
SmtpTaskAbort	Abort the specified asynchronous task
SmtpTaskDone	Determine if an asynchronous task has completed
SmtpTaskResume	Resume execution of an asynchronous task
SmtpTaskSuspend	Suspend execution of an asynchronous task
SmtpTaskWait	Wait for an asynchronous task to complete
SmtpUninitialize	Terminate use of the library by the application
SmtpVerifyAddress	Verify that the specified address is valid
SmtpWrite	Write data to the server

SmtpAddRecipient Function

```
INT WINAPI SmtpAddRecipient(  
    HCLIENT hClient,  
    LPCTSTR lpszAddress  
);
```

The **SmtpAddRecipient** function adds the specified address to the recipient list for the current message. This function should be called once for each recipient.

Parameters

hClient

Handle to the client session.

lpszAddress

Points to a string which specifies the address to be added to the recipient list.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtpCloseMessage](#), [SmtpCreateMessage](#), [SmtpExpandAddress](#), [SmtpVerifyAddress](#)

SmtpAppendMessage Function

```
INT WINAPI SmtpAppendMessage(  
    HCLIENT hClient,  
    LPVOID lpvMessage,  
    DWORD dwMessageSize,  
    DWORD dwOptions  
);
```

The **SmtpAppendMessage** function writes the contents of a specified file or buffer to the data stream, appending it to the current message contents.

Parameters

hClient

Handle to the client session.

lpvMessage

Pointer to a buffer which contains the message data to be appended, or a pointer to the name of the file which contains the data to be written to the data stream. The use of this parameter depends on the value of the *dwOptions* parameter.

dwMessageSize

An unsigned integer which specifies the length of the message in bytes.

dwOptions

Specifies the source of the message data that will be written to the data stream; it may be one of the following values:

Constant	Description
SMTP_MESSAGE_MEMORY	The <i>lpvMessage</i> parameter specifies a pointer to an array of characters. If the value of <i>dwMessageSize</i> is zero, then it is assumed to be a pointer to a string.
SMTP_MESSAGE_HGLOBAL	The <i>lpvMessage</i> parameter specifies an HGLOBAL which contains the data to be written to the data stream. If the value of the <i>dwMessageSize</i> parameter is zero, then the data is assumed to be null-terminated.
SMTP_MESSAGE_FILE	The <i>lpvMessage</i> parameter specifies a pointer to a string which contains the name of a file. The file is opened and the contents of the file are written to the data stream. The value of the <i>dwMessageSize</i> parameter is ignored when this option is specified.
SMTP_MESSAGE_CLIPBOARD	The <i>lpvMessage</i> and <i>dwMessageSize</i> parameters are ignored. The current contents of the clipboard are written to the data stream.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpAppendMessage** function is used to append the contents of a memory buffer, file or

the system clipboard to the current message that is being composed for delivery. To send a complete RFC 822 formatted message, refer to the **SmtpSendMessage** function.

This function will cause the current thread to block until the complete message has been written, a timeout occurs or the operation is canceled. During the transfer, the SMTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **SmtpEnableEvents**, or by registering a callback function using the **SmtpRegisterEvent** function.

To determine the current status of a transfer while it is in progress, use the **SmtpGetTransferStatus** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtpCloseMessage](#), [SmtpCreateMessage](#), [SmtpSendMessage](#)

SmtpAsyncConnect Function

```
HCLIENT WINAPI SmtpAsyncConnect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPCTSTR lpszLocalName,  
    LPSECURITYCREDENTIALS lpCredentials,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **SmtpAsyncConnect** function is used to establish a connection with the server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **SmtpConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 25. An alternative port is 587, which is commonly used by authenticated clients to submit messages for delivery. For secure connections, the default port number is 465. If the secure port number is specified, an implicit SSL/TLS connection will be established by default.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SMTP_OPTION_NONE	No additional options are specified when establishing a connection with the server. A standard, non-secure connection will be used and the client will not attempt to use extended (ESMTP) features of the protocol. Note that if the mail server requires authentication, the SMTP_OPTION_EXTENDED option must be specified.

SMTP_OPTION_EXTENDED	Extended SMTP commands should be used if possible. This option enables features such as authentication and delivery status notification. If this option is not specified, the library will not attempt to use any extended features. This option is automatically enabled if the connection is established on port 587 because submitting messages for delivery using this port typically requires client authentication.
SMTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
SMTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
SMTP_OPTION_SECURE	This option specifies that a secure connection should be established with the server and requires that the server support either the SSL or TLS protocol. This option is the same as specifying SMTP_OPTION_SECURE_EXPLICIT, which initiates the secure session using the STARTTLS command.
SMTP_OPTION_SECURE_EXPLICIT	This option specifies the client should attempt to establish a secure connection with the server using the STARTTLS command. Note that the server must support secure connections using either the SSL or TLS protocol.
SMTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using either the SSL or TLS protocol, and the secure session must be negotiated immediately after the connection has been established.
SMTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
SMTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This

	option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
SMTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpzLocalName

A pointer to a string which specifies the domain name of the local host. This parameter can be NULL or point to an empty string, in which case the local domain name is determined automatically from the system configuration. This parameter should be used if the mail server only accepts messages from a client that identifies itself using a specific domain name.

lpCredentials

A pointer to a [SECURITYCREDENTIALS](#) structure which is used to establish the client credentials for a secure connection to the server. The function **SmtplibCreateSecurityCredentials** can be used to create this structure if necessary. If a standard non-secure connection is being established, or client credentials are not required by the server, this parameter can be NULL.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous socket events that occur. If this parameter is NULL, a blocking connection is established with the server.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the *hEventWnd* parameter is NULL, this parameter must be specified as WM_NULL.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **SmtplibGetLastError**.

Remarks

The *lpzLocalName* argument should only specify a domain name if it is absolutely necessary. In most cases, it is preferable to pass this parameter as NULL or an empty string and allow the library to automatically determine the correct domain name to use. Providing an invalid domain name may cause the mail server to reject the connection.

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description

SMTP_EVENT_CONNECT	The connection to the server has completed. The high word of the IParam parameter should be checked, since this notification message will be posted if an error has occurred.
SMTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
SMTP_EVENT_READ	Data is available to read by the client. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the calling process is in asynchronous mode.
SMTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
SMTP_EVENT_TIMEOUT	The client has timed out while waiting for a response from the server. Note that under some circumstances this event can be generated for a non-blocking connection, such as when the client is establishing a secure connection.
SMTP_EVENT_CANCEL	The client has canceled the current operation.
SMTP_EVENT_COMMAND	The client has processed a command that was sent to the server. The result code and result string can be used to determine if the response to the command. The high word of the IParam parameter should be checked, since this notification message will also be posed if the command cannot be executed.
SMTP_EVENT_PROGRESS	This event notification is sent periodically during lengthy blocking operations, such as retrieving a complete message from the server.

To cancel asynchronous notification and return the client to a blocking mode, use the **SmtplibDisableEvents** function.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **SmtplibAttachThread** function.

Specifying the SMTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtpAuthenticate](#), [SmtpConnect](#), [SmtpDisconnect](#), [SmtpInitialize](#), [SmtpUninitialize](#)

SmtpAsyncSendMessage Function

```
UINT WINAPI SmtpAsyncSendMessage(  
    HCLIENT hClient,  
    LPCTSTR lpszFrom,  
    LPCTSTR lpszRecipient,  
    LPVOID lpvMessage,  
    DWORD dwMessageSize,  
    DWORD dwOptions,  
    SMTPEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

Send the contents of a file or memory buffer to the specified recipients.

Parameters

hClient

Handle to the client session.

lpszFrom

Pointer to a string which specifies the email address of the sender.

lpszRecipient

Pointer to a string which specifies the recipient of the message. Multiple recipients may be specified by separating each address with a comma.

lpvMessage

Pointer to a buffer which contains the message to be delivered, or a pointer to the name of the file which contains the data to be written to the data stream. The use of this parameter depends on the value of the *dwOptions* parameter.

dwMessageSize

An unsigned integer which specifies the length of the message in bytes.

dwOptions

Specifies the source of the message data that will be written to the data stream; it may be one of the following values:

Constant	Description
SMTP_MESSAGE_MEMORY	The <i>lpvMessage</i> parameter specifies a pointer to an array of characters. If the value of <i>dwMessageSize</i> is zero, then it is assumed to be a pointer to a string.
SMTP_MESSAGE_HGLOBAL	The <i>lpvMessage</i> parameter specifies an HGLOBAL which contains the data to be written to the data stream. If the value of the <i>dwMessageSize</i> parameter is zero, then the data is assumed to be null-terminated.
SMTP_MESSAGE_FILE	The <i>lpvMessage</i> parameter specifies a pointer to a string which contains the name of a file. The file is opened and the contents of the file are written to the data stream. The value of the <i>dwMessageSize</i> parameter is ignored when this option is specified.
SMTP_MESSAGE_CLIPBOARD	The <i>lpvMessage</i> and <i>dwMessageSize</i> parameters are

ignored. The current contents of the clipboard are written to the data stream.
--

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **SmtplibEventProc** callback function. This parameter may be NULL if you do not wish to implement an event handler.

dwParam

A user-defined integer value that is passed to the callback function. This parameter is ignored if the *lpEventProc* parameter is NULL.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **SmtplibGetLastError** function.

Remarks

The **SmtplibAsyncSendMessage** function is used to send the contents of a memory buffer, file or the system clipboard to the specified recipients. This function is similar to the **SmtplibSendMessage** function, however it uses a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the message is being submitted to mail server for delivery.

Because this function works asynchronously, it is important that the memory allocated for the message is not released before the asynchronous task completes. If you provide a buffer that is allocated on the stack, ensure that your code does not return from the function while the message is being submitted. This can be achieved by calling the **SmtplibTaskWait** function or periodically calling the **SmtplibTaskDone** function to determine if the background task has completed. If you wish to return from the calling function immediately, then you must dynamically allocate memory for the *lpvMessage* parameter on the heap and free that memory after the task has completed.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the SMTP_EVENT_CONNECT event after the connection has been established, and the SMTP_EVENT_DISCONNECT event after the message has been submitted. This enables your application to know when the message is being submitted to the mail server, and immediately before the worker thread is terminated. The worker thread creates a secondary connection to the server with its own session handle. This ensures that the asynchronous operation will not interfere with the current client session. Your application can interact with this background worker thread using the client handle that is passed to the event handler.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtplib10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtplibAsyncSubmitMessage](#), [SmtplibSendMessage](#), [SmtplibSubmitMessage](#), [SmtplibTaskWait](#)

SmtpAsyncSubmitMessage Function

```
UINT WINAPI SmtpAsyncSubmitMessage(  
    LPSMTPSERVER lpServer,  
    LPSMTPMESSAGE lpMessage,  
    SMTPEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

The **SmtpSubmitMessage** function composes and submits a message for delivery to the specified mail server.

Parameters

lpServer

A pointer to an **SMTPSERVER** structure that contains information about the mail server that the message will be submitted to for delivery. This parameter cannot be NULL and the structure members must be properly initialized prior to calling this function.

lpMessage

A pointer to an **SMTPMESSAGE** structure that contains information about the message, including the sender, recipients and the body of the message. This parameter cannot be NULL and the structure members must be property initialized prior to calling this function.

lpEventProc

A pointer to the procedure-instance address of an application defined callback function. For more information about event handling and the callback function, see the description of the **SmtpEventProc** callback function. If this parameter is NULL, event notification is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the *lpEventProc* parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the function fails, the return value is zero. To get extended error information, call the **SmtpGetLastError** function.

Remarks

The **SmtpAsyncSubmitMessage** function provides a high-level interface that enables an application to send an email message with a single function call. This function is similar to the **SmtpSubmitMessage** function; however, it uses a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the message is being submitted to the mail server.

The **SMTPSERVER** and **SMTPMESSAGE** structures are used to provide the function with information about the mail server that will accept the message and the contents of the message itself. Note that this function does not require a client session handle, and therefore it is not required that you call the **SmtpConnect** function prior to calling this function.

If the address of an event handler is provided to this function, it is guaranteed that the handler will be invoked with the SMTP_EVENT_CONNECT event after the connection has been established, and the SMTP_EVENT_DISCONNECT event after the message has been submitted. This enables your application to know when the message is being submitted to the mail server, and

immediately before the worker thread is terminated. The worker thread creates a secondary connection to the server with its own session handle. This ensures that the asynchronous operation will not interfere with the current client session. Your application can interact with this background worker thread using the client handle that is passed to the event handler.

Example

```
SMTPSERVER mailServer;
ZeroMemory(&mailServer, sizeof(mailServer));
mailServer.lpszHostName = _T("smtp.gmail.com");
mailServer.nHostPort = SMTP_PORT_SUBMIT;
mailServer.lpszUserName = m_strSender;
mailServer.lpszPassword = m_strPassword;
mailServer.dwOptions = SMTP_OPTION_SECURE;

SMTPMESSAGE mailMessage;
ZeroMemory(&mailMessage, sizeof(mailMessage));
mailMessage.lpszFrom = m_strSender;
mailMessage.lpszTo = m_strRecipients;
mailMessage.lpszSubject = m_strSubject;
mailMessage.lpszText = m_strMessage;

UINT nTaskId = SmtAsyncSubmitMessage(&mailServer, &mailMessage, NULL, 0);

if (nTaskId != 0)
{
    DWORD dwError = NO_ERROR;
    DWORD dwElapsed = 0;

    // Wait for the message to be submitted
    SmtTaskWait(nTaskId, INFINITE, &dwElapsed, &dwError);

    if (dwError == NO_ERROR)
        _tprintf(_T("SmtAsyncSubmitMessage was successful\n"));
}
else
{
    DWORD dwError = SmtGetLastError();
    _tprintf(_T("SmtSubmitMessage failed with error 0x%08lx\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtAsyncSendMessage](#), [SmtEventProc](#), [SmtSendMessage](#), [SmtSubmitMessage](#), [SmtTaskWait](#), [SMTPMESSAGE](#), [SMTPSERVER](#)

SmtpAttachThread Function

```
DWORD WINAPI SmtpAttachThread(  
    HCLIENT hClient  
    DWORD dwThreadId  
);
```

The **SmtpAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **SmtpAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **SmtpAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **SmtpCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **SmtpUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmtpv10.lib

See Also

[SmtpCancel](#), [SmtpAsyncConnect](#), [SmtpConnect](#), [SmtpDisconnect](#), [SmtpUninitialize](#)

SmtpAuthenticate Function

```
INT WINAPI SmtpAuthenticate(  
    HCLIENT hClient,  
    UINT nAuthType,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword  
);
```

The **SmtpAuthenticate** function provides client authentication information to the server.

Parameters

hClient

The handle to the client session.

nAuthType

An integer value which specifies which method the library should use to authenticate the client session. This parameter should be set to one of the following values:

Constant	Description
SMTP_AUTH_LOGIN	The client will authenticate using the AUTH LOGIN command. This encodes the username and password, however the credentials are not encrypted and it is recommended you use a secure connection. This is the default method accepted by most mail servers and is the preferred authentication type for most clients.
SMTP_AUTH_PLAIN	The client will authenticate using the AUTH PLAIN command. This encodes the username and password, however the credentials are not encrypted and it is recommended you use a secure connection. The server must support the PLAIN Simple Authentication and Security Layer (SASL) mechanism as defined in RFC 4616.
SMTP_AUTH_XOAUTH2	The client will authenticate using the AUTH XOAUTH2 command. This authentication method does not require the user password, instead the <i>lpszPassword</i> parameter must specify the OAuth 2.0 bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this function will fail.
SMTP_AUTH_BEARER	The client will authenticate using the AUTH OAUTHBEARER command as defined in RFC 7628. This authentication method does not require the user password, instead the <i>lpszPassword</i> parameter must specify the OAuth 2.0 bearer token issued by the service provider. The application must provide a valid access token which has not expired, or this function will fail.

lpszUserName

A null terminated string which specifies the account name for the user authorized to send mail through the server.

lpszPassword

A null terminated string which specifies the password to be used when authenticating the current client session. If you are using the SMTP_AUTH_XOAUTH2 or SMTP_AUTH_BEARER authentication methods, this parameter is not a password, instead it specifies the OAuth 2.0 access token provided by the mail service.

Return Value

If the function succeeds, the return value is the command result code. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtplibGetLastError**.

Remarks

To submit a mail message for delivery, virtually all public mail servers require clients to authenticate and will only accept messages from authorized users. In some cases, they may also require the sender email address match the account being used to authenticate the session. It is also typical for most public mail servers to reject authentication attempts over a standard (non-secure) connection. You should always use a secure connection whenever possible.

All authentication methods require the mail server to support the standard service extensions for authentication as specified in the RFC 4954. The server must support the ESMTP protocol extensions and the AUTH command. A user name and password are required for authentication. If you wish to authenticate without a user password, you must use one of the OAuth 2.0 authentication methods.

The default authentication method is SMTP_AUTH_LOGIN and this is accepted by most mail servers. It is common for mail servers to allow the SMTP_AUTH_PLAIN method as well, however it is recommended you explicitly check whether the server supports the desired authentication method by calling the **SmtplibGetExtendedOptions** function. If you attempt to use an authentication method which is not supported by the server, this function will fail and the last error code will be set to ST_ERROR_INVALID_AUTHENTICATION_TYPE.

You should only use an OAuth 2.0 authentication method if you understand the process of how to request the access token. Obtaining an access token requires registering your application with the mail service provider (e.g.: Microsoft or Google), getting a unique client ID associated with your application and then requesting the access token using the appropriate scope for the service. Obtaining the initial token will typically involve interactive confirmation on the part of the user, requiring they grant permission to your application to access their mail account.

The SMTP_AUTH_XOAUTH2 and SMTP_AUTH_BEARER authentication methods are similar, but they are not interchangeable. Both use an OAuth 2.0 bearer token to authenticate the client session, but they differ in how the token is presented to the server. It is currently preferable to use the XOAUTH2 method because it is more widely available and some service providers do not yet support the OAUTHBEARER method.

Your application should not store an OAuth 2.0 bearer token for later use. They have a relatively short lifespan, typically about an hour, and are designed to be used with that session. You should specify offline access as part of the OAuth 2.0 scope if necessary and store the refresh token provided by the service. The refresh token has a much longer validity period and can be used to obtain a new bearer token when needed.

Example

```
BOOL bExtended = FALSE;  
DWORD dwOptions = 0;
```

```
// Determine which extended options and authentication methods
```

```

// are supported by this server

bExtended = SntpGetExtendedOptions(hClient, &dwOptions);

if (bUseBearerToken)
{
    if (bExtended && (dwOptions & SMTP_EXTOPT_XOAUTH2))
    {
        INT nResult = SntpAuthenticate(hClient, SMTP_AUTH_XOAUTH2, lpszUserName,
lpszBearerToken);

        if (nResult == SMTP_ERROR)
        {
            // An error occurred during authentication; when using an
            // OAuth 2.0 bearer token, this typically means that the token
            // has expired and must be refreshed
            return;
        }
    }
    else
    {
        // The server does not support XOAUTH2
        return;
    }
}
else
{
    if (bExtended && (dwOptions & SMTP_EXTOPT_AUTHLOGIN))
    {
        INT nResult = SntpAuthenticate(hClient, SMTP_AUTH_LOGIN, lpszUserName,
lpszPassword);

        if (nResult == SMTP_ERROR)
        {
            // An error occurred during authentication
            return;
        }
    }
    else
    {
        // The server does not support AUTH LOGIN
        return;
    }
}
}

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SntpConnect](#), [SntpGetExtendedOptions](#)

SmtpCancel Function

```
INT WINAPI SmtpCancel(  
    HCLIENT hClient  
);
```

The **SmtpCancel** function cancels any outstanding blocking operation in the client, causing the blocking function to fail. The application may then retry the operation or terminate the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

When the **SmtpCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtpIsBlocking](#)

SmtpCloseMessage Function

```
INT WINAPI SmtpCloseMessage(  
    HCLIENT hClient  
);
```

The **SmtpCloseMessage** function ends the composition of the current message. The server then queues the message for delivery to each recipient specified by the client.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpCloseMessage** function should be called after all of the message data has been written to the data stream.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtpAppendMessage](#), [SmtpCreateMessage](#), [SmtpWrite](#)

SmtpCommand Function

```
INT WINAPI SmtpCommand(  
    HCLIENT hClient,  
    LPCTSTR lpszCommand,  
    LPCTSTR lpszParameter  
);
```

The **SmtpCommand** function sends a command to the server and returns the result code back to the caller. This function is typically used for site-specific commands not directly supported by the API.

Parameters

hClient

Handle to the client session.

lpszCommand

The command which will be executed by the server.

lpszParameter

An optional command parameter. If the command requires more than one parameter, then they should be combined into a single string, with a space separating each parameter. If the command does not accept any parameters, this value may be NULL.

Return Value

If the command was successful, the function returns the result code. If the command failed, the function returns SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

A list of valid commands can be found in the technical specification for the protocol. Many servers will list supported commands when the HELP command is used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtpGetResultCode](#), [SmtpGetResultString](#)

SmtpConnect Function

```
HCLIENT WINAPI SmtpConnect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPCTSTR lpszLocalName,  
    LPSECURITYCREDENTIALS lpCredentials  
);
```

The **SmtpConnect** function is used to establish a connection with the server.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 25. An alternative port is 587, which is commonly used by authenticated clients to submit messages for delivery. For implicit SSL connections, the default port number is 465.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SMTP_OPTION_NONE	No additional options are specified when establishing a connection with the server. A standard, non-secure connection will be used and the client will not attempt to use extended (ESMTP) features of the protocol. Note that if the mail server requires authentication, the SMTP_OPTION_EXTENDED option must be specified.
SMTP_OPTION_EXTENDED	Extended SMTP commands should be used if possible. This option enables features such as authentication and delivery status notification. If this option is not specified, the library will not attempt to use any extended features. This option is automatically enabled if the connection is established on port 587 because submitting messages for delivery using this port typically requires client authentication.
SMTP_OPTION_TUNNEL	This option specifies that a tunneled TCP

	connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
SMTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
SMTP_OPTION_SECURE	This option specifies that a secure connection should be established with the server and requires that the server support either the SSL or TLS protocol. The client will initiate the secure session using the STARTTLS command.
SMTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using either the SSL or TLS protocol, and the secure session must be negotiated immediately after the connection has been established.
SMTP_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
SMTP_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
SMTP_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpzLocalName

A pointer to a string which specifies the domain name of the local host. This parameter can be NULL or point to an empty string, in which case the local domain name is determined

automatically from the system configuration. This parameter should be used if the mail server only accepts messages from a client that identifies itself using a specific domain name.

lpCredentials

A pointer to a [SECURITYCREDENTIALS](#) structure which is used to establish the client credentials for a secure connection to the server. The function **SsmtpCreateSecurityCredentials** can be used to create this structure if necessary. If a standard non-secure connection is being established, or client credentials are not required by the server, this parameter can be NULL.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **SsmtpGetLastError**.

Remarks

To prevent this function from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **SsmtpConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

The *lpzLocalName* argument should only point to a specific domain name if it is absolutely necessary. In most cases, it is preferable to pass this parameter as NULL or an empty string and allow the library to automatically determine the correct domain name to use. Providing an invalid domain name may cause the mail server to reject the connection.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **SsmtpAttachThread** function.

Specifying the SMTP_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SsmtpAuthenticate](#), [SsmtpDisconnect](#), [SsmtpInitialize](#), [SsmtpUninitialize](#)

SmtpCreateMessage Function

```
INT WINAPI SmtpCreateMessage(  
    HCLIENT hClient  
    LPCTSTR lpszSender,  
    DWORD dwMessageSize,  
    DWORD dwReserved  
);
```

The **SmtpCreateMessage** function creates a new message for delivery.

Parameters

hClient

Handle to the client session.

lpszSender

A pointer to a string which specifies the email address of the user sending the message. This typically corresponds to the address in the From header of the message, but it is not required that they be the same.

dwMessageSize

An unsigned integer which specifies the size of the message in bytes. If the size of the message is unknown, this value should be zero. This parameter is ignored if the server does not support extended features. If the message size is larger than what the server will accept, this function will fail. Most Internet Service Providers impose a limit on the size of an email message, typically between 5 and 10 megabytes.

dwReserved

A reserved parameter. This value should be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpCreateMessage** function begins the composition of a new message to be submitted to the mail server for delivery. There are several steps that must be followed when dynamically composing a message for delivery:

1. Call the **SmtpCreateMessage** function to begin the message composition. The sender email address should generally be the same address as the one used in the "From" header field in the message.
2. Call the **SmtpAddRecipient** function for each recipient of the message. These addresses are typically specified in the "To" and "Cc" header fields in the message. Additional addresses may also be provided which are not specified in the email message itself. This is how one or more blind carbon copies of a message is delivered. Most servers have a limit on the total number of recipients that may be specified for a single message. This limit is usually around 100 addresses.
3. Call the **SmtpWrite** function to write the contents of the message to the data stream. The application may also choose to use the **SmtpAppendMessage** function to write out a large amount of message data, or write the contents of a file to the data stream.

4. Call the **SmtplibCloseMessage** function to close the message and submit it to the mail server for delivery.

For applications that do not need to dynamically compose the message and already have the message contents stored in a file or memory buffer, the **SmtplibSendMessage** function is the preferred method of submitting a message for delivery.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtplib10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtplibAddRecipient](#), [SmtplibAppendMessage](#), [SmtplibCloseMessage](#), [SmtplibSendMessage](#), [SmtplibWrite](#)

SmtpCreateSecurityCredentials Function

```
BOOL WINAPI SmtpCreateSecurityCredentials(  
    DWORD dwProtocol,  
    DWORD dwOptions,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName,  
    LPVOID lpvReserved,  
    LPSECURITYCREDENTIALS* lppCredentials  
);
```

The **SmtpCreateSecurityCredentials** function creates a **SECURITYCREDENTIALS** structure.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is

	supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A pointer to a string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A pointer to a string which specifies the certificate owner's password. A value of NULL specifies

that no password is required. This parameter is only used if a PKCS12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A pointer to a string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

lpvReserved

Pointer reserved for future use. Set it to NULL when using this function.

lppCredentials

Pointer to an [LPSECURITYCREDENTIALS](#) pointer. The memory for the credentials structure will be allocated by this function and must be released by calling the **SsmtpDeleteSecurityCredentials** function when it is no longer needed. The pointer value must be set to NULL before the function is called. It is important to note that this is a pointer to a pointer variable, not a pointer to the SECURITYCREDENTIALS structure itself.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **SsmtpGetLastError**.

Remarks

The structure that is created by this function may be used as client credentials when establishing a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can

be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **SmtAsyncConnect** or **SmtConnect** functions.

Example

```
LPSECURITYCREDENTIALS lpSecCred = NULL;
SmtCreateSecurityCredentials(SEcurity_PROTOCOL_DEFAULT,
    0,
    NULL,
    NULL,
    lpszCertStore,
    lpszCertName,
    NULL,
    &lpSecCred);

hClient = SmtConnect(lpszHostName,
    SMTP_PORT_SECURE,
    SMTP_TIMEOUT,
    SMTP_OPTION_EXTENDED | SMTP_OPTION_SECURE,
    lpSecCred);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtAsyncConnect](#), [SmtConnect](#), [SmtDeleteSecurityCredentials](#), [SmtGetSecurityInformation](#), [SECURITYCREDENTIALS](#)

SmtDeleteSecurityCredentials Function

```
VOID WINAPI SmtDeleteSecurityCredentials(  
    LPSECURITYCREDENTIALS* lppCredentials  
);
```

The **SmtDeleteSecurityCredentials** function deletes an existing **SECURITYCREDENTIALS** structure.

Parameters

lppCredentials

Pointer to an **LPSECURITYCREDENTIALS** pointer. On exit from the function, the pointer will be NULL.

Return Value

None.

Example

```
if (lpSecCred)  
    SmtDeleteSecurityCredentials(&lpSecCred);  
  
SmtUninitialize();
```

Remarks

This function can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtCreateSecurityCredentials](#), [SmtUninitialize](#)

SmtplibDisableEvents Function

```
INT WINAPI SmtplibDisableEvents(  
    HCLIENT hClient  
);
```

The **SmtplibDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtplibGetLastError**.

Remarks

The **SmtplibDisableEvents** function is used to disable event message posting for the specified client session. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this function has been called.

This function is automatically called if the client has event notification enabled, and the **SmtplibDisconnect** function is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a client handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtplib10.lib

See Also

[SmtplibEnableEvents](#), [SmtplibRegisterEvent](#)

SmtplibDisableTrace Function

```
BOOL WINAPI SmtplibDisableTrace();
```

The **SmtplibDisableTrace** function disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

To get extended error information, call **SmtplibGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtplib10.lib

See Also

[SmtplibEnableTrace](#)

SmtpDisconnect Function

```
INT WINAPI SmtpDisconnect(  
    HCLIENT hClient  
);
```

The **SmtpDisconnect** function terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtpAsyncConnect](#), [SmtpConnect](#), [SmtpUninitialize](#)

Smtplib.EnableEvents Function

```
INT WINAPI Smtplib.EnableEvents(  
    HCLIENT hClient,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **Smtplib.EnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **Smtplib.RegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the window which will receive the client notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.

uEventMsg

The message that is received when a client event occurs. To avoid conflict with standard Windows messages, this value must be greater than WM_USER (1024) or an error will be returned. If the *hEventWnd* parameter is NULL, this value should be WM_NULL.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **Smtplib.GetLastError**.

Remarks

The **Smtplib.EnableEvents** function is used to request that notification messages be posted to the specified window whenever a client event occurs. This allows an application to monitor the status of different client operations, such as a file transfer. The client must create a window message handler, which processes the various events. The *wParam* argument will contain the client handle, the low word of the *lParam* argument will contain the event identifier, and the high word will contain any error code. If no error has occurred, the high word will have a value of zero. The following events may be generated:

Constant	Description
SMTP_EVENT_CONNECT	The connection to the server has completed. The high word of the lParam parameter should be checked, since this notification message will be posted if an error has occurred.
SMTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
SMTP_EVENT_READ	Data is available to read by the client. No additional messages will be posted until the client has read at least some of the data.

	This event is only generated if the calling process is in asynchronous mode.
SMTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
SMTP_EVENT_TIMEOUT	The client has timed out while waiting for a response from the server. Note that under some circumstances this event can be generated for a non-blocking connection, such as when the client is establishing a secure connection.
SMTP_EVENT_CANCEL	The client has canceled the current operation.
SMTP_EVENT_COMMAND	The client has processed a command that was sent to the server. The result code and result string can be used to determine if the response to the command. The high word of the IParam parameter should be checked, since this notification message will also be posed if the command cannot be executed.
SMTP_EVENT_PROGRESS	This event notification is sent periodically during lengthy blocking operations, such as retrieving a complete message from the server.

It is not required that the client be placed in asynchronous (non-blocking) mode in order to receive event notifications, except for the connect, disconnect, read and write events. To disable event notification, call the **SmtplibDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtplib10.lib

See Also

[SmtplibDisableEvents](#), [SmtplibRegisterEvent](#)

SntpEnableTrace Function

```
BOOL WINAPI SntpEnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **SntpEnableTrace** function enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

Name of the trace log file. If this parameter is NULL or empty, the file CSTRACE.LOG is used. The directory for CSTRACE.LOG is given by the TEMP environment variable, if it is defined; otherwise, the directory given by the TMP environment variable is used, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Constant	Description
0	TRACE_DEFAULT	All function calls and return values are written to the trace file. The actual data being sent or received will not be logged. This is the default value.
1	TRACE_ERROR	Only those function calls which fail are recorded in the trace file. Those errors which are not fatal and only indicate a warning will not be logged.
2	TRACE_WARNING	Only those function calls which fail, or return values which indicate a warning, are recorded in the trace file.
4	TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **SntpGetLastError**.

Remarks

When trace logging is enabled, the logfile is opened, appended to and closed for each socket function call. Using the same logfile name, you can do the same in your application to add additional information to the logfile if needed. This can provide an application-level context for the entries made by the library. Make sure that the logfile is closed after the data has been written.

The TRACE_HEXDUMP option can produce very large logfiles, since all data that is being sent and received by the application is logged. To reduce the size of the file, you can enable and disable logging around limited sections of code that you wish to analyze.

All of the SocketTools networking components that use the Windows Sockets API support logging. If you are using multiple components, you only need to enable tracing once in your application or once per thread in a multithreaded application.

To redistribute an application that includes logging functionality, the **cstrcv10.dll** library must be included as part of the installation package. This library provides the trace logging features, and if it is not available the **SmtpEnableTrace** function will fail. Note that the trace logging library is a standard Windows DLL and does not need to be registered, it only needs to be redistributed with your application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtpDisableTrace](#)

SmtpEnumTasks Function

```
INT WINAPI SmtpEnumTasks(  
    UINT * lpTasks,  
    INT nMaxTasks,  
    DWORD dwOptions  
);
```

Return a list of active, suspended or finished asynchronous tasks.

Parameters

lpTasks

A pointer to an array of unsigned integer values that will contain unique task identifiers when the function returns. If this parameter is NULL, the function will return the number of tasks.

nMaxTasks

An integer value that specifies the maximum number of task identifiers that may be copied into the *lpTasks* array. If the *lpTasks* parameter is NULL, this value must be zero.

dwOptions

An unsigned integer that specifies the type of asynchronous tasks that may be returned by this function. It may be a combination of the following values:

Constant	Description
SMTP_TASK_DEFAULT	The list of asynchronous task IDs should include both active and suspended tasks. This option is the same as specifying both the SMTP_TASK_ACTIVE and SMTP_TASK_SUSPENDED options.
SMTP_TASK_ACTIVE	The list of asynchronous task IDs should include those tasks which are currently active. An active task represents a background connection to a server that is in the process of performing the requested action, such as uploading or downloading a file.
SMTP_TASK_SUSPENDED	The list of asynchronous task IDs should include those tasks which have been suspended. A suspended task represents a background connection that has been established, but the worker thread is not scheduled for execution.
SMTP_TASK_FINISHED	The list of asynchronous task IDs should include those tasks which have completed recently.

Return Value

If the function is successful, the return value is the number of task identifiers copied into the provided array. If there are no tasks which match the requested criteria, the return value is zero. A return value of SMTP_ERROR indicates an error has occurred. To get extended error information, call the **SmtpGetLastError** function.

Remarks

The **SmtpEnumTasks** function can be used to obtain a list of numeric identifiers that represent the asynchronous tasks that have been started or those that have completed. These task IDs are used by other functions to reference the background worker thread that has been created and

obtain status information for the task. For example, the **SntpTaskDone** function can be used to determine if a particular task has completed, and the **SntpTaskWait** function can be used to wait for a task to complete and return an error status code if the background operation failed.

There is an internal limit of 128 asynchronous tasks per process that may be active at any one time. When a task completes, the status information about that task is maintained for period of time after the task has completed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

See Also

[SntpTaskDone](#), [SntpTaskResume](#), [SntpTaskSuspend](#), [SntpTaskWait](#)

SmtpEventProc Function

```
VOID CALLBACK SmtpEventProc(  
    HCLIENT hClient,  
    UINT nEvent,  
    DWORD dwError,  
    DWORD_PTR dwParam  
);
```

The **SmtpEventProc** function is an application-defined callback function that processes events generated by the client.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **SmtpRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **SmtpRegisterEvent** function. This callback function is also used by asynchronous tasks to notify the application when the task has started and completed. The **SmtpEventProc** function is a placeholder for the application-defined function name.

If the callback function is invoked by an asynchronous task, it will execute in the context of the worker thread that is managing the client session. You must ensure that any access to global or static variables are synchronized, otherwise the results may be unpredictable. It is recommended that you do not declare any static variables within the callback function itself.

If the application has a graphical user interface, you should never attempt to directly modify a UI control from within the callback function for an asynchronous task. Controls should only be modified by the same UI thread that created their window. One common approach to resolve this issue is to post a user-defined message to the main window to signal that the user interface needs to be updated. The message handler would then process the user-defined message and update the user interface as needed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmtpv10.lib

See Also

[SmtpDisableEvents](#), [SmtpEnableEvents](#), [SmtpFreezeEvents](#), [SmtpRegisterEvent](#)

SmtpExpandAddress Function

```
INT WINAPI SmtpExpandAddress(  
    HCLIENT hClient,  
    LPCTSTR lpszMailingList,  
    LPTSTR lpszAddresses,  
    INT nMaxLength  
);
```

The **SmtpExpandAddress** function expands the specified mailing list, returning the membership of that list.

Parameters

hClient

Handle to the client session.

lpszMailingList

Points to a string which specifies the mailing list that the server should expand into full addresses.

lpszAddresses

Points to a buffer that the expanded addresses will be copied into.

nMaxLength

Maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpExpandAddress** function requests that the server expand the specified email address. Typically this is used to expand aliases which refer to a mailing list, returning all of the members of that list. A server may not support this command, or may restrict its usage. An application should not depend on the ability to expand addresses.

This function cannot be called while a mail message is being composed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtpAddRecipient](#), [SmtpVerifyAddress](#)

SmtplibFreezeEvents Function

```
INT WINAPI SmtplibFreezeEvents(  
    HCLIENT hClient,  
    BOOL bFreeze  
);
```

The **SmtplibFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtplibGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four events of the same type occur, once event handling is resumed only one of those events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtplib10.lib

See Also

[SmtplibDisableEvents](#), [SmtplibEnableEvents](#), [SmtplibRegisterEvent](#)

SmtpGetCurrentDate Function

```
INT WINAPI SmtpGetCurrentDate(  
    LPTSTR lpszDateString,  
    INT nMaxLength  
);
```

The **SmtpGetCurrentDate** function copies the current date and time to the specified buffer in a format that is commonly used in mail messages. This date format should be used in all date-related fields in the message header.

Parameters

lpszDateString

Pointer to a string buffer that will contain the current date and time when the function returns.

nMaxLength

The maximum number of characters that can be copied into the string buffer.

Return Values

If the function succeeds, the return value is the number of characters copied into the buffer, not including the null-terminator. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The date value that is returned is adjusted for the local timezone.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtpCloseMessage](#), [SmtpCreateMessage](#)

SmtpGetDeliveryOptions Function

```
BOOL WINAPI SmtpGetDeliveryOptions(  
    HCLIENT hClient,  
    LPDWORD LpdwOptions  
);
```

The **SmtpGetDeliveryOptions** function returns the delivery status notification options for the current session.

Parameters

hClient

Handle to the client session.

lpdwOptions

Address of a variable that will be set to the current delivery options. This bitmask is created by combining one or more of the following values with a bitwise Or operator:

Constant	Description
SMTP_NOTIFY_NEVER	Never return information about the success or failure of the message delivery process.
SMTP_NOTIFY_SUCCESS	Return a message to the sender if the message has been successfully delivered to the recipient's mail server.
SMTP_NOTIFY_FAILURE	Return a message to the sender if the message could not be delivered to the recipient's mail server.
SMTP_NOTIFY_DELAY	Return a message to the sender if delivery of the message was delayed.
SMTP_RETURN_HEADERS	Return only the message headers to the sender.
SMTP_RETURN_MESSAGE	Return the complete message headers and body to the sender.

Return Values

If the function succeeds, the return value is a non-zero value. If the function fails, the return value is zero. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpGetDeliveryOptions** function returns the current delivery options for the client session. Note that delivery options are only available on those mail servers which support delivery status notification (DSN) using the extended SMTP protocol. The client must connect specifying SMTP_OPTION_EXTENDED in order to use extended server options.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

See Also

[SmtpAsyncConnect](#), [SmtpConnect](#), [SmtpGetExtendedOptions](#), [SmtpSetDeliveryOptions](#)

SmtpGetErrorString Function

```
INT WINAPI SmtpGetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT nMaxLength  
);
```

The **SmtpGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **SmtpGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description should be returned. If this value is zero, then the description of the last error will be returned. If the last error code is zero, indicating no error, then this function will return zero.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. It is recommended that this buffer be at least 128 characters in length. If a NULL pointer is specified, then no message will be returned but the function will return the length of the error string, not including the terminating null byte.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is 0, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmtpv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtpGetLastError](#), [SmtpGetResultCode](#), [SmtpGetResultString](#), [SmtpSetLastError](#)

SmtpGetExtendedOptions Function

```
BOOL WINAPI SmtpGetExtendedOptions(  
    HCLIENT hClient,  
    LPDWORD lpdwOptions  
);
```

The **SmtpGetExtendedOptions** function returns the extended server options for the current session.

Parameters

hClient

Handle to the client session.

lpdwOptions

Address of a variable that will be set to the current server options. This bitmask is created by combining one or more of the following values with a bitwise Or operator:

Constant	Description
SMTP_EXTOPT_EXPN	The server supports address expansion using the EXPN command. The SmtpExpandAddress function can be used to expand addresses, which typically returns the email addresses associated with a mailing list. Most public mail servers restrict or disable this functionality because it can present a security risk. If a server does permit the use of the command, it is often limited to specific authorized users.
SMTP_EXTOPT_VRFY	The server supports verification of addresses using the VRFY command. The SmtpVerifyAddress function can be used to verify addresses. Most public mail servers restrict the ability for clients to verify email addresses to prevent potential abuse. If a server does permit the use of the command, it is often limited to specific authorized users.
SMTP_EXTOPT_DSN	The server supports delivery status notification (DSN) which allows the sender to be notified when a message has been delivered, or when an error occurs during the delivery process. The SmtpSetDeliveryOptions function can be used to specify the delivery options to be used in the current session.
SMTP_EXTOPT_SIZE	The server supports the use of the SIZE parameter, which enables the client to determine the maximum message size that may be delivered through the server. Most public mail servers impose a limit of on the total size of a message, including any encoded attachments.
SMTP_EXTOPT_ETRN	The server supports the use of the ETRN command, instructing the server to start processing its message queues for a specific host. Most public mail servers do not support this capability and its use has been deprecated.
SMTP_EXTOPT_8BITMIME	The server supports the delivery of messages that contain

	characters with the high bit set. Most servers support this option, however it is recommended that you encode any message text which contains non-ASCII characters to ensure the broadest compatibility with other servers and clients.
SMTP_EXTOPT_STARTTLS	The server supports explicit TLS sessions. This extended option is used internally to determine how secure connections should be established, and if a secure connection can be made using the standard submission port.
SMTP_EXTOPT_UTF8	The server supports UTF-8 encoding in email addresses and the message envelope. Not all mail servers will have this extended capability enabled, and applications should not depend on being able to provide internationalized user and domain names unless this option bitflag has been set.

In addition, there are extended options which specify the authentication methods supported by the server. A server will typically support multiple authentication methods and may be one or more of the following values:

Constant	Description
SMTP_EXTOPT_AUTHLOGIN	The server supports client authentication using the AUTH LOGIN command. This is the default authentication method and is supported by most mail servers. The user name and password are encoded in a specific format, but are not encrypted. The client should use a secure connection whenever possible.
SMTP_EXTOPT_AUTHPLAIN	The server supports client authentication using the AUTH PLAIN command. The use name and password are encoded in a specific format, but are not encrypted. If a server supports this authentication method, it is very likely it also supports AUTH LOGIN. It is recommended you use only use AUTH PLAIN authentication if the server does not support AUTH LOGIN.
SMTP_EXTOPT_XOAUTH2	The server supports client authentication using AUTH XOAUTH2 command. Instead of a password, an OAuth 2.0 bearer token is used to authenticate the user which previously authorized access to the mail server using their account information. The connection must be secure to use this authentication method.
SMTP_EXTOPT_BEARER	The server supports client authentication using AUTH OAUTHBEARER command as specified in RFC 7628. Instead of a password, an OAuth 2.0 bearer token is used to authenticate the user which previously authorized access to the mail server using their account information. The connection must be secure to use this authentication method.

Return Value

If the function succeeds, the return value is a non-zero value. If the function fails, the return value is zero. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpGetExtendedOptions** function returns the extended options supported by the server. The use of extended options requires that the server support the ESMTP protocol, and that the client connect using the SMTP_OPTION_EXTENDED option.

You should check these options prior to calling **SmtpAuthenticate** to determine which authentication methods are acceptable to the server. If you wish to use an OAuth 2.0 bearer token, always check to make sure either the SMTP_EXTOPT_XOAUTH2 or SMTP_EXTOPT_BEARER bitflags are set in the options value returned by this function.

Example

```
BOOL bExtended = FALSE;
DWORD dwOptions = 0;

// Determine which extended options and authentication methods
// are supported by this server

bExtended = SmtpGetExtendedOptions(hClient, &dwOptions);

if (bExtended && (dwOptions & SMTP_EXTOPT_XOAUTH2))
{
    INT nResult = SmtpAuthenticate(hClient, SMTP_AUTH_XOAUTH2, lpszUserName,
    lpszBearerToken);

    if (nResult == SMTP_ERROR)
    {
        // An error occurred during authentication; when using an
        // OAuth 2.0 bearer token, this typically means that the token
        // has expired and must be refreshed
        return;
    }
}
else
{
    // The server does not support XOAUTH2
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmtpv10.lib

See Also

[SmtpAsyncConnect](#), [SmtpAuthenticate](#), [SmtpConnect](#), [SmtpGetDeliveryOptions](#), [SmtpSetDeliveryOptions](#)

Smtplib GetLastError Function

```
DWORD WINAPI Smtplib GetLastError();
```

Parameters

None.

Return Value

The return value is the calling thread's last error code value. Functions set this value by calling the **Smtplib SetLastError** function. The return value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

You should call the **Smtplib GetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **Smtplib SetLastError(0)** when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value error code such as FALSE, NULL, INVALID_CLIENT or SMTP_ERROR. Those functions which call **Smtplib SetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtplib10.lib

See Also

[Smtplib GetLastErrorString](#), [Smtplib SetLastError](#)

SmtpGetResultCode Function

```
INT WINAPI SmtpGetResultCode(  
    HCLIENT hClient  
);
```

The **SmtpGetResultCode** function reads the result code returned by the server in response to a command. The result code is an integer value, and indicates if the operation succeeded or failed.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the result code. If the function fails, it returns SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The following result codes may be returned by the SMTP server:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

See Also

[SmtpCommand](#), [SmtpGetResultString](#)

SmtpGetResultString Function

```
INT WINAPI SmtpGetResultString(  
    HCLIENT hClient,  
    LPTSTR lpszResult,  
    INT nMaxLength  
);
```

The **SmtpGetResultString** function returns the last message sent by the server along with the result code.

Parameters

hClient

Handle to the client session.

lpszResult

A pointer to the buffer that will contain the result string returned by the server.

nMaxLength

The maximum number of characters that may be copied into the result string buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the result string. If a value of zero is returned, this means that no result string was sent by the server. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpGetResultString** function is most useful when an error occurs because the server will typically include a brief description of the cause of the error. This can then be parsed by the application or displayed to the user. The result string is updated each time the client sends a command to the server and then calls **SmtpGetResultCode** to obtain the result code for the operation.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtpCommand](#), [SmtpGetResultCode](#)

SmtplibGetSecurityInformation Function

```
BOOL WINAPI SmtplibGetSecurityInformation(  
    HCLIENT hClient,  
    LPSECURITYINFO lpSecurityInfo  
);
```

The **SmtplibGetSecurityInformation** function returns security protocol, encryption and certificate information about the current client connection.

Parameters

hClient

Handle to the client session.

lpSecurityInfo

A pointer to a [SECURITYINFO](#) structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **SmtplibGetLastError**.

Remarks

This function is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value `SECURITY_PROTOCOL_NONE`.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;  
  
securityInfo.dwSize = sizeof(SECURITYINFO);  
if (SmtplibGetSecurityInformation(hClient, &securityInfo))  
{  
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)  
    {  
        MessageBox(NULL, _T("The connection is not secure"),  
                    _T("Connection"), MB_OK);  
    }  
    else  
    {  
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)  
        {  
            MessageBox(NULL, _T("The connection is secure"),  
                        _T("Connection"), MB_OK);  
        }  
        else  
        {  
            MessageBox(NULL, _T("The server certificate not valid"),  
                        _T("Connection"), MB_OK);  
        }  
    }  
}
```

```
}  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtAsyncConnect](#), [SmtConnect](#), [SmtDisconnect](#), [SECURITYINFO](#)

SmtpGetStatus Function

```
INT WINAPI SmtpGetStatus(  
    HCLIENT hClient  
);
```

The **SmtpGetStatus** function returns the current status of the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpGetStatus** function returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Constant	Description
1	SMTP_STATUS_IDLE	The client is current idle and not sending or receiving data.
2	SMTP_STATUS_CONNECT	The client is establishing a connection with the server.
3	SMTP_STATUS_READ	The client is reading data from the server.
4	SMTP_STATUS_WRITE	The client is writing data to the server.
5	SMTP_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

See Also

[SmtpIsBlocking](#), [SmtpIsConnected](#), [SmtpIsReadable](#), [SmtpIsWritable](#)

SntpGetTaskError Function

```
DWORD WINAPI SntpGetTaskError(  
    UINT nTaskId  
);
```

Return the last error code for the specified asynchronous task.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has completed successfully, this function returns a value of zero. A non-zero return value indicates an error has occurred.

Remarks

The **SntpGetTaskError** function returns the last error code associated with the specified asynchronous task. If the task completed successfully, the return value will be zero. If the task is still active, the function will return the error ST_ERROR_TASK_ACTIVE. If the task has been suspended, the function will return ST_ERROR_TASK_SUSPENDED. Any other value indicates that the task completed, but the operation has failed and the error code will specify the cause of the failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

See Also

[SntpTaskAbort](#), [SntpTaskResume](#), [SntpTaskSuspend](#), [SntpTaskWait](#)

SmtpGetTaskId Function

```
UINT WINAPI SmtpGetTaskId(  
    HCLIENT hClient  
);
```

Return the asynchronous task identifier associated with the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is non-zero integer value that specifies a unique asynchronous task identifier. If the client handle is not associated with an asynchronous task, the function will return a value of zero.

Remarks

The **SmtpGetTaskId** function will return the task ID that is associated with a client session. This is a unique unsigned integer value that references the worker thread that was created to manage the asynchronous client session. This function should only be called within an event handler that is invoked by a background task that has been started using a function such as **SmtpAsyncGetFile**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csmtpv10.lib`

See Also

[SmtpTaskAbort](#), [SmtpTaskDone](#), [SmtpTaskResume](#), [SmtpTaskSuspend](#), [SmtpTaskWait](#)

SmtplibTimeout Function

```
INT WINAPI SmtplibTimeout(  
    HCLIENT hClient  
);
```

The **SmtplibTimeout** function returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtplibGetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtplib10.lib

See Also

[SmtplibSetTimeout](#)

SmtpGetTransferStatus Function

```
INT WINAPI SmtpGetTransferStatus(  
    HCLIENT hClient,  
    LPSMTPTRANSFERSTATUS lpStatus  
);
```

The **SmtpGetTransferStatus** function returns information about the message being submitted to the mail server.

Parameters

hClient

Handle to the client session.

lpStatus

A pointer to an [SMTPTRANSFERSTATUS](#) structure which contains information about the status of the message being submitted for delivery.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpGetTransferStatus** function returns information about the current message being submitted, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no message currently being submitted, this function will return the status of the last successful submission made by the client.

In a multithreaded application, any thread in the current process may call this function to obtain the status of a submission for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtpEnableEvents](#), [SmtpGetStatus](#), [SmtpRegisterEvent](#)

SmtplibInitialize Function

```
BOOL WINAPI SmtplibInitialize(  
    LPCTSTR lpszLicenseKey,  
    LPINITDATA lpData  
);
```

The **SmtplibInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpData

Pointer to an INITDATA data structure. This parameter may be NULL if the initialization data for the library is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **SmtplibGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must insure that this function is called with a valid license key.

If the *lpData* argument is specified, it must point to an INITDATA structure which has been initialized by setting the *dwSize* member to the size of the structure. All other structure members should be set to zero. If the function is successful, then the INITDATA structure will be filled with identifying information about the library.

Although it is only required that **SmtplibInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **SmtplibUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **SmtplibInitialize** or **SmtplibUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtplib10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

SmtplibBlocking Function

```
BOOL WINAPI SmtplibBlocking(  
    HCLIENT hClient  
);
```

The **SmtplibBlocking** function is used to determine if the client is currently performing a blocking operation.

Parameters

hClient

Handle to the client session.

Return Value

If the client is performing a blocking operation, the function returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the function returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking function may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **SmtplibBlocking** function can be used to determine if the client is already blocked, and if so, take some other action such as warning the user that they must wait for the operation to complete.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtplib10.lib

See Also

[SmtplibCancel](#), [SmtplibConnected](#), [SmtplibReadable](#), [SmtplibWritable](#)

SmtplibConnected Function

```
BOOL WINAPI SmtplibConnected(  
    HCLIENT hClient  
);
```

The **SmtplibConnected** function is used to determine if the client is currently connected to a server.

Parameters

hClient

Handle to the client session.

Return Value

If the client is connected to a server, the function returns a non-zero value. If the client is not connected, or the client handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtplib10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtplibBlocking](#), [SmtplibReadable](#), [SmtplibWritable](#)

SmtpIsReadable Function

```
BOOL WINAPI SmtpIsReadable(  
    HCLIENT hClient,  
    INT nTimeout,  
    LPDWORD lpdwAvail  
);
```

The **SmtpIsReadable** function is used to determine if data is available to be read from the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the function returns a non-zero value. If the client cannot read any data, the function returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

See Also

[SmtpGetStatus](#), [SmtpIsBlocking](#), [SmtpIsConnected](#), [SmtpIsWritable](#), [SmtpWrite](#)

SmtpIsWritable Function

```
BOOL WINAPI SmtpIsWritable(  
    HCLIENT hClient,  
    INT nTimeout  
);
```

The **SmtpIsWritable** function is used to determine if data can be written to the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the function returns a non-zero value. If the client cannot write any data, the function returns zero.

Remarks

Although this function can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmtpv10.lib

See Also

[SmtpGetStatus](#), [SmtpIsBlocking](#), [SmtpIsConnected](#), [SmtpIsReadable](#), [SmtpWrite](#)

SmtplibRegisterEvent Function

```
INT WINAPI SmtplibRegisterEvent(  
    HCLIENT hClient,  
    UINT nEvent,  
    SMTPEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

The **SmtplibRegisterEvent** function registers a callback function for the specified event.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Constant	Description
SMTP_EVENT_CONNECT	The connection to the server has completed.
SMTP_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
SMTP_EVENT_READ	Data is available to read by the client. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the calling process is in asynchronous mode.
SMTP_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
SMTP_EVENT_TIMEOUT	The client has timed out while waiting for a response from the server. Note that under some circumstances this event can be generated for a non-blocking connection, such as when the client is establishing a secure connection.
SMTP_EVENT_CANCEL	The client has canceled the current operation.
SMTP_EVENT_COMMAND	The client has processed a command that was sent to the server. The result code and result string can be used to determine if the response to the command.
SMTP_EVENT_PROGRESS	This event notification is sent periodically during lengthy blocking operations, such as retrieving a complete message from the server.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **SmtplibEventProc** callback

function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtplibGetLastError**.

Remarks

The **SmtplibRegisterEvent** function associates a callback function with a specific event. The event handler is an **SmtplibEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

This function is typically used to register an event handler that is invoked while a message is being submitted to the server for delivery. The SMTP_EVENT_PROGRESS event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one SMTP_EVENT_PROGRESS notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtplib10.lib

See Also

[SmtplibDisableEvents](#), [SmtplibEnableEvents](#), [SmtplibEventProc](#), [SmtplibFreezeEvents](#)

Smtplib.Reset Function

```
INT WINAPI Smtplib.Reset(  
    HCLIENT hClient  
);
```

The **Smtplib.Reset** function resets the client state and resynchronizes with the server. This function is typically called after an unexpected error has occurred, or an operation has been canceled.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **Smtplib.GetLastError**.

Remarks

The client cannot be reset while in a blocked state.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtplib10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[Smtplib.Cancel](#), [Smtplib.CloseMessage](#), [Smtplib.CreateMessage](#), [Smtplib.IsBlocking](#)

SmtpSendMessage Function

```
INT WINAPI SmtpSendMessage(  
    HCLIENT hClient,  
    LPCTSTR lpszFrom,  
    LPCTSTR lpszRecipient,  
    LPVOID lpvMessage,  
    DWORD dwMessageSize,  
    DWORD dwOptions  
);
```

The **SmtpSendMessage** function sends the contents of a file or memory buffer to the specified recipients.

Parameters

hClient

Handle to the client session.

lpszFrom

Pointer to a string which specifies the email address of the sender.

lpszRecipient

Pointer to a string which specifies the recipient of the message. Multiple recipients may be specified by separating each address with a comma.

lpvMessage

Pointer to a buffer which contains the message to be delivered, or a pointer to the name of the file which contains the data to be written to the data stream. The use of this parameter depends on the value of the **dwOptions** parameter.

dwMessageSize

An unsigned integer which specifies the length of the message in bytes.

dwOptions

Specifies the source of the message data that will be written to the data stream; it may be one of the following values:

Constant	Description
SMTP_MESSAGE_MEMORY	The lpvMessage parameter specifies a pointer to an array of characters. If the value of dwMessageSize is zero, then it is assumed to be a pointer to a string.
SMTP_MESSAGE_HGLOBAL	The lpvMessage parameter specifies an HGLOBAL which contains the data to be written to the data stream. If the value of the dwMessageSize parameter is zero, then the data is assumed to be null-terminated.
SMTP_MESSAGE_FILE	The lpvMessage parameter specifies a pointer to a string which contains the name of a file. The file is opened and the contents of the file are written to the data stream. The value of the dwMessageSize parameter is ignored when this option is specified.
SMTP_MESSAGE_CLIPBOARD	The lpvMessage and dwMessageSize parameters are ignored. The current contents of the clipboard are

written to the data stream.

Return Value

If the function succeeds, the return value is the result code from the server. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtplibGetLastError**.

Remarks

The **SmtplibSendMessage** function is used to send the contents of a memory buffer, file or the system clipboard to the specified recipients. The message must be in the standard format as described in RFC 822 or a MIME multipart message. The MIME API can be used to compose and export a message in the correct format.

This protocol is only concerned with the delivery of a message and not its contents. Header fields in the message are not parsed to determine the recipients. This recipient parameter should be a concatenation of all recipients, including carbon copies and blind carbon copies, with each address separated with a comma.

This function will cause the current thread to block until the complete message has been delivered, a timeout occurs or the operation is canceled. During the transfer, the SMTP_EVENT_PROGRESS event will be periodically fired, enabling the application to update any user interface controls. Event notification must be enabled, either by calling **SmtplibEnableEvents**, or by registering a callback function using the **SmtplibRegisterEvent** function.

To determine the current status of the transaction while it is in progress, use the **SmtplibGetTransferStatus** function.

An alternative approach to creating a message without using the MIME API is the **SmtplibSubmitMessage** function. It accepts two structure parameters which define the message contents and the connection information for the mail server. This enables the application to compose the message and submit it for delivery in a single function call.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtplib10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtplibAppendMessage](#), [SmtplibCloseMessage](#), [SmtplibGetTransferStatus](#), [SmtplibSubmitMessage](#)

SmtpSetDeliveryOptions Function

```
BOOL WINAPI SmtpSetDeliveryOptions(  
    HCLIENT hClient,  
    DWORD dwOptions  
);
```

The **SmtpSetDeliveryOptions** function sets the delivery status notification options for the current session.

Parameters

hClient

Handle to the client session.

dwOptions

A bitmask that defines the current delivery options. This value is created by combining one or more of the following constants with a bitwise Or operator:

Constant	Description
SMTP_NOTIFY_NEVER	Never return information about the success or failure of the message delivery process.
SMTP_NOTIFY_SUCCESS	Return a message to the sender if the message has been successfully delivered to the recipient's mail server.
SMTP_NOTIFY_FAILURE	Return a message to the sender if the message could not be delivered to the recipient's mail server.
SMTP_NOTIFY_DELAY	Return a message to the sender if delivery of the message was delayed.
SMTP_RETURN_HEADERS	Return only the message headers to the sender.
SMTP_RETURN_MESSAGE	Return the complete message headers and body to the sender.

Return Value

If the function succeeds, the return value is a non-zero value. If the function fails, the return value is zero. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpSetDeliveryOptions** function sets the current delivery options for the client session. Note that delivery options are only available on those mail servers which support delivery status notification (DSN) using the extended SMTP protocol. The client must connect specifying SMTP_OPTION_EXTENDED in order to use extended server options.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

SmtpSetLastError Function

```
VOID WINAPI SmtpSetLastError(  
    DWORD dwErrorCode  
);
```

The **SmtpSetLastError** function sets the last error code for the current thread.

Parameters

dwErrorCode

Specifies the last error code for the caller. A value of zero clears the last error code.

Return Value

None.

Remarks

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value error code such as FALSE, NULL, INVALID_CLIENT or SMTP_ERROR. Those functions which call **SmtpSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **SmtpGetLastError** function. The use of **SmtpGetLastError** is optional. An application can call it to find out the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

See Also

[SmtpGetErrorString](#), [SmtpGetLastError](#)

SmtpSetTimeout Function

```
INT WINAPI SmtpSetTimeout(  
    HCLIENT hClient,  
    INT nTimeout  
);
```

The **SmtpSetTimeout** function sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

See Also

[SmtpGetTimeout](#)

SmtpSubmitMessage Function

```
INT WINAPI SmtpSubmitMessage(  
    LPSMTPSERVER LpServer,  
    LPSMTPMESSAGE LpMessage,  
    SMTPEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

The **SmtpSubmitMessage** function composes and submits a message for delivery to the specified mail server.

Parameters

lpServer

A pointer to an **SMTPSERVER** structure that contains information about the mail server that the message will be submitted to for delivery. This parameter cannot be NULL and the structure members must be properly initialized prior to calling this function.

lpMessage

A pointer to an **SMTPMESSAGE** structure that contains information about the message, including the sender, recipients and the body of the message. This parameter cannot be NULL and the structure members must be property initialized prior to calling this function.

lpEventProc

A pointer to the procedure-instance address of an application defined callback function. For more information about event handling and the callback function, see the description of the **SmtpEventProc** callback function. If this parameter is NULL, event notification is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the *lpEventProc* parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpSubmitMessage** function provides a high-level interface that enables an application to send an email message with a single function call. The **SMTPSERVER** and **SMTPMESSAGE** structures are used to provide the function with information about the mail server that will accept the message and the contents of the message itself. Note that this function does not require a client session handle, and therefore it is not required that you call the **SmtpConnect** function prior to calling this function.

If you need to specify additional custom headers in the message that is submitted for delivery, you should use the **SmtpSubmitMessageEx** function. That version of the function uses an extended version of the message structure which will allow you to define custom headers to be included in the message.

This function will cause the calling thread to block until the message has been submitted for delivery, an error occurs or the connection to the mail server times out. If an event handler is specified, then the callback function will be periodically invoked as the message is being sent. For large messages, the SMTP_EVENT_PROGRESS event can be used to monitor the submission

process and update the user interface. The **SmtplibGetTransferStatus** function can be used within the callback function to obtain information about the current status of the submission.

Example

```
SMTPSERVER mailServer;
ZeroMemory(&mailServer, sizeof(mailServer));
mailServer.lpszHostName = _T("smtp.gmail.com");
mailServer.nHostPort = SMTP_PORT_SUBMIT;
mailServer.lpszUserName = m_strSender;
mailServer.lpszPassword = m_strPassword;
mailServer.dwOptions = SMTP_OPTION_SECURE;

SMTPMESSAGE mailMessage;
ZeroMemory(&mailMessage, sizeof(mailMessage));
mailMessage.lpszFrom = m_strSender;
mailMessage.lpszTo = m_strRecipients;
mailMessage.lpszSubject = m_strSubject;
mailMessage.lpszText = m_strMessage;

INT nResult = SmtplibSubmitMessage(&mailServer, &mailMessage, NULL, 0);

if (nResult != SMTP_ERROR)
    _tprintf(_T("SmtplibSubmitMessage was successful\n"));
else
{
    DWORD dwError = SmtplibGetLastError();
    _tprintf(_T("SmtplibSubmitMessage failed with error 0x%08lx\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtplib10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtplibEventProc](#), [SmtplibSendMessage](#), [SmtplibSubmitMessageEx](#), [SMTPMESSAGE](#), [SMTPSERVER](#)

SmtpSubmitMessageEx Function

```
INT WINAPI SmtpSubmitMessageEx(  
    LPSMTPSERVER LpServer,  
    LPSMTPMESSAGEEX LpMessageEx,  
    DWORD dwReserved,  
    SMTPEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

The **SmtpSubmitMessageEx** function composes and submits a message for delivery to the specified mail server.

Parameters

lpServer

A pointer to an **SMTPSERVER** structure that contains information about the mail server that the message will be submitted to for delivery. This parameter cannot be NULL and the structure members must be properly initialized prior to calling this function.

lpMessageEx

A pointer to an **SMTPMESSAGEEX** structure that contains information about the message, including the sender, recipients and the body of the message. This parameter cannot be NULL and the structure members must be property initialized prior to calling this function.

dwReserved

An integer value that is reserved for future use. This value must be zero or the function will fail.

lpEventProc

A pointer to the procedure-instance address of an application defined callback function. For more information about event handling and the callback function, see the description of the **SmtpEventProc** callback function. If this parameter is NULL, event notification is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the *lpEventProc* parameter is NULL, this value should be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The **SmtpSubmitMessageEx** function provides a high-level interface that enables an application to send an email message with a single function call. The **SMTPSERVER** and **SMTPMESSAGEEX** structures are used to provide the function with information about the mail server that will accept the message and the contents of the message itself. Note that this function does not require a client session handle, and therefore it is not required that you call the **SmtpConnect** function prior to calling this function.

This function will cause the calling thread to block until the message has been submitted for delivery, an error occurs or the connection to the mail server times out. If an event handler is specified, then the callback function will be periodically invoked as the message is being sent. For large messages, the SMTP_EVENT_PROGRESS event can be used to monitor the submission process and update the user interface. The **SmtpGetTransferStatus** function can be used within

the callback function to obtain information about the current status of the submission.

Example

```
SMTPSERVER mailServer;
ZeroMemory(&mailServer, sizeof(mailServer));
mailServer.lpszHostName = _T("smtp.gmail.com");
mailServer.nHostPort = SMTP_PORT_SUBMIT;
mailServer.lpszUserName = m_strSender;
mailServer.lpszPassword = m_strPassword;
mailServer.dwOptions = SMTP_OPTION_SECURE;

SMTPMESSAGEEX mailMessageEx;
ZeroMemory(&mailMessageEx, sizeof(mailMessageEx));
mailMessageEx.dwSize = sizeof(mailMessageEx);
mailMessageEx.lpszFrom = m_strSender;
mailMessageEx.lpszTo = m_strRecipients;
mailMessageEx.lpszSubject = m_strSubject;
mailMessageEx.lpszText = m_strMessage;

INT nResult = SmtplibSubmitMessageEx(&mailServer, &mailMessageEx, 0, NULL, 0);

if (nResult != SMTP_ERROR)
    _tprintf(_T("SmtplibSubmitMessageEx was successful\n"));
else
{
    DWORD dwError = SmtplibGetLastError();
    _tprintf(_T("SmtplibSubmitMessageEx failed with error 0x%08lx\n"), dwError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtplib10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtplibEventProc](#), [SmtplibSendMessage](#), [SmtplibSubmitMessage](#), [SMTPMESSAGEEX](#), [SMTPSERVER](#)

SmtpTaskAbort Function

```
BOOL WINAPI SmtpTaskAbort(  
    UINT nTaskId,  
    DWORD dwMilliseconds  
);
```

Abort the specified asynchronous task.

Parameters

nTaskId

The task identifier.

dwMilliseconds

An unsigned integer that specifies the number of milliseconds to wait for the background task to abort.

Return Value

If the function succeeds and the worker thread has terminated, the return value is non-zero. A return value of zero indicates that the worker thread is still running or an error has occurred. To get extended error information, call the **SmtpGetLastError** function.

Remarks

The **SmtpTaskAbort** function signals the background worker thread associated with the task ID to abort the current operation and terminate as soon as possible. If the *dwMilliseconds* parameter has a value of zero, the function returns immediately after the background thread has been signaled. If the *dwMilliseconds* parameter is non-zero, the function will wait that amount of time for the background thread to terminate.

This function should never be called from within the event handler for an asynchronous task because it can cause the process to deadlock. To abort a file transfer within an event handler, use the **SmtpCancel** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csmtpv10.lib

See Also

[SmtpTaskDone](#), [SmtpTaskResume](#), [SmtpTaskSuspend](#), [SmtpTaskWait](#)

SmtpTaskDone Function

```
BOOL WINAPI SmtpTaskDone(  
    UINT nTaskId  
);
```

Determine if an asynchronous task has completed.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has completed, this function returns a non-zero value. A return value of zero indicates that the worker thread is still running or an error has occurred. To get extended error information, call the **SmtpGetLastError** function.

Remarks

The **SmtpTaskDone** function is used to determine if the specified asynchronous task has completed. If you use this function to poll the status of a background task from within the main UI thread, you must ensure that Windows messages are processed so that the application remains responsive to the end-user. To check if a background transfer has completed, it is recommended that you use a timer to periodically call this function rather than calling it repeatedly within a loop.

To determine if the task completed successfully, the **SmtpGetTaskError** function will return the last error code associated with the task. A return value of zero indicates success, while a non-zero return value specifies an error code that indicates the cause of the failure. The last error code for the task can also be retrieved using the **SmtpTaskWait** function, which causes the application to wait for the asynchronous task to complete.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

See Also

[SmtpGetTaskError](#), [SmtpTaskAbort](#), [SmtpTaskResume](#), [SmtpTaskSuspend](#), [SmtpTaskWait](#)

SmtptaskResume Function

```
BOOL WINAPI SmtptaskResume(  
    UINT nTaskId  
);
```

Resume execution of an asynchronous task.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has resumed, this function returns a non-zero value. A return value of zero indicates that an error has occurred. To get extended error information, call the **SmtptaskGetLastError** function.

Remarks

The **SmtptaskResume** function resumes execution of the background worker thread that was previously suspended using the **SmtptaskSuspend** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

See Also

[SmtptaskAbort](#), [SmtptaskDone](#), [SmtptaskSuspend](#), [SmtptaskWait](#)

SmtptaskSuspend Function

```
BOOL WINAPI SmtptaskSuspend(  
    UINT nTaskId  
);
```

Suspend execution of an asynchronous task.

Parameters

nTaskId

The task identifier.

Return Value

If the asynchronous task has resumed, this function returns a non-zero value. A return value of zero indicates that an error has occurred. To get extended error information, call the **SmtptaskGetLastError** function.

Remarks

The **SmtptaskSuspend** function will suspend execution of the background worker thread associated with the task. Once the task has been suspended, it will no longer be scheduled for execution, however the client session will remain active and the task may be resumed using the **SmtptaskResume** function. Note that if a task is suspended for a long period of time, the background operation may fail because it has exceeded the timeout period imposed by the server.

This function should never be called from within the event handler for an asynchronous task because it can cause the process to deadlock.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

See Also

[SmtptaskAbort](#), [SmtptaskDone](#), [SmtptaskResume](#), [SmtptaskWait](#)

SntpTaskWait Function

```
BOOL WINAPI SntpTaskWait(  
    UINT nTaskId,  
    DWORD dwMilliseconds,  
    DWORD dwReserved,  
    LPDWORD lpdwElapsed,  
    LPDWORD lpdwError  
);
```

Wait for an asynchronous task to complete.

Parameters

nTaskId

The task identifier.

dwMilliseconds

An unsigned integer that specifies the number of milliseconds to wait for the background task to complete.

dwReserved

An unsigned integer reserved for future use. This value should always be zero.

lpdwElapsed

A pointer to an unsigned integer that will contain elapsed time in milliseconds when the function returns. If this information is not required, this parameter may be NULL.

lpdwError

A pointer to an unsigned integer that will contain the error code associated with the completed task. If this information is not required, this parameter may be NULL.

Return Value

If the function succeeds and the worker thread has terminated, the return value is non-zero. A return value of zero indicates that the worker thread is still running or an error has occurred. To get extended error information, call the **SntpGetLastError** function.

Remarks

The **SntpTaskWait** function waits for the specified task to complete. If the task is active and the *dwMilliseconds* parameter is non-zero, this function will cause the current working thread to block until the task completes or the amount of time exceeds the number of milliseconds specified by the caller. If the *dwMilliseconds* parameter is zero, then this function will poll the status of the task and return immediately to the caller.

If the specified task has already completed at the time this function is called, the function will return immediately without causing the current thread to block. If the *lpdwElapsed* parameter is not NULL, it will contain the number of milliseconds that it took for the task to complete. If the *lpdwError* parameter is not NULL, it will contain the last error code value that was set by the worker thread before it terminated. If this value is zero, that means that the background operation was successful and no error occurred. A non-zero value will indicate that the background operation has failed.

You should not call this function from the main UI thread with a long timeout period to wait for a background task to complete. Windows messages will not be processed while this function is blocked waiting for the background task to complete, and this can cause your application to

appear non-responsive to the end-user. If you have a GUI application and you need to periodically check to see if a task has completed, create a timer to periodically call the **SmtptaskDone** function. When it returns a non-zero value (indicating that the task has completed), you can safely call **SmtptaskWait** to obtain the elapsed time and last error code without blocking the current thread.

This function should never be called from within the event handler for an asynchronous task because it can cause the process to deadlock.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

See Also

[SmtptaskDone](#), [SmtptaskResume](#), [SmtptaskSuspend](#), [SmtptaskWait](#)

SmtpUninitialize Function

```
VOID WINAPI SmtpUninitialize();
```

The **SmtpUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **SmtpInitialize** call before it can call any of the other library functions. When it has completed the use of library, the application must call **SmtpUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There must be a call to **SmtpUninitialize** for every successful call to **SmtpInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csmtpv10.lib

See Also

[SmtpAsyncConnect](#), [SmtpConnect](#), [SmtpDisconnect](#), [SmtpInitialize](#)

SmtplibVerifyAddress Function

```
INT WINAPI SmtplibVerifyAddress(  
    HCLIENT hClient,  
    LPCTSTR lpszAddress,  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);
```

The **SmtplibVerifyAddress** function verifies the specified address is valid.

Parameters

hClient

Handle to the client session.

lpszAddress

Points to a string which specifies the address that the server should verify.

lpszBuffer

Points to a buffer that the verified address will be copied into.

nMaxLength

Maximum number of characters that may be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the server result code. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtplibGetLastError**.

Remarks

The **SmtplibVerifyAddress** function requests that the server verify the specified email address. Typically this is used to verify that a recipient address is valid, and return a fully qualified email address for that recipient. A server may not support this command, or may restrict its usage. An application should not depend on the ability to verify addresses.

This function cannot be called while a mail message is being composed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtplib10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmtplibAddRecipient](#), [SmtplibExpandAddress](#)

SmtpWrite Function

```
INT WINAPI SmtpWrite(  
    HCLIENT hClient,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **SmtpWrite** function sends the specified number of bytes to the server.

Parameters

hClient

Handle to the client session.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is SMTP_ERROR. To get extended error information, call **SmtpGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the SMTP_EVENT_WRITE asynchronous notification message before it resumes sending data.

If the **SmtpWrite** function is used to send the message contents to the server, the application must first call the **SmtpCreateMessage** function to specify the sender and the length of the message, followed by one or more calls to the **SmtpAddRecipient** function to specify each recipient of the message. When all of the message text has been submitted to the server, the application must call the **SmtpCloseMessage** function.

The message text is filtered by the **SmtpWrite** function, and it will automatically normalize end-of-line character sequences to ensure the message meets the protocol requirements. The message itself must be in a standard RFC 822 or multi-part MIME message format, or the server may reject the message. Binary data, such as file attachments, should always be encoded. The MIME API can be used to compose and export a message in the correct format, which can then be submitted to the server.

It is recommended that most applications use the **SmtpSendMessage** function to submit the message for delivery.

An alternative approach to creating a message without using the MIME API is the **SmtpSubmitMessage** function. It accepts two structure parameters which define the message contents and the connection information for the mail server. This enables the application to compose the message and submit it for delivery in a single function call.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csmtpv10.lib

See Also

[SmtpAddRecipient](#), [SmtpAppendMessage](#), [SmtpCloseMessage](#), [SmtpCreateMessage](#),
[SmtpSendMessage](#), [SmtpSubmitMessage](#)

Simple Message Transfer Protocol Data Structures

- INITDATA
- SECURITYCREDENTIALS
- SECURITYINFO
- SMTPMESSAGE
- SMTPMESSAGEEX
- SMTPSERVER
- SMTPTRANSFERSTATUS
- SYSTEMTIME

INITDATA Structure

This structure contains information about the SocketTools library when the initialization function returns.

```
typedef struct _INITDATA
{
    DWORD        dwSize;
    DWORD        dwVersionMajor;
    DWORD        dwVersionMinor;
    DWORD        dwVersionBuild;
    DWORD        dwOptions;
    DWORD_PTR    dwReserved1;
    DWORD_PTR    dwReserved2;
    TCHAR        szDescription[128];
} INITDATA, *LPINITDATA;
```

Members

dwSize

Size of this structure. This structure member must be set to the size of the structure prior to calling the initialization function. Failure to do so will cause the function to fail.

dwVersionMajor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the major version number for the library.

dwVersionMinor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the minor version number for the library.

dwVersionBuild

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the build number for the library.

dwOptions

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved1

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved2

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

szDescription

A null terminated string which describes the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

SECURITYCREDENTIALS Structure

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on

	what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that

will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A pointer to a null terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A pointer to a null terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A pointer to a null terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A pointer to a null terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
------------	-------------

CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
----	---

MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
----	--

ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.
------	--

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

A pointer to a null terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used for SSH connections

and should always be NULL when establishing a secure connection using SSL or TLS.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* member should be set to CREDENTIAL_STORE_FILENAME and the *lpzCertStore* member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS#12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the *lpzPassword* member or the library will be unable to access the certificate.

Note that the *lpzUserName* and *lpzPassword* members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

The TLS 1.1 and TLS 1.2 protocols are only supported on Windows 7, Windows Server 2008 R2 and later versions of the platform. If these options are specified and the application is running on Windows XP or Windows Vista, the protocol version will be downgraded to TLS 1.0 for backwards compatibility with those platforms.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

SECURITYINFO Structure

This structure contains information about a secure connection that has been established.

```
typedef struct _SECURITYINFO
{
    DWORD        dwSize;
    DWORD        dwProtocol;
    DWORD        dwCipher;
    DWORD        dwCipherStrength;
    DWORD        dwHash;
    DWORD        dwHashStrength;
    DWORD        dwKeyExchange;
    DWORD        dwCertStatus;
    SYSTEMTIME   stCertIssued;
    SYSTEMTIME   stCertExpires;
    LPCTSTR      lpszCertIssuer;
    LPCTSTR      lpszCertSubject;
    LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to **sizeof(SECURITYINFO)** prior to passing the address of this structure to the function. Note that if this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the protocol that was selected to establish the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The

	<p>correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.</p>
SECURITY_PROTOCOL_TLS10	<p>The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS11	<p>The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS12	<p>The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.</p>

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_CIPHER_RC2	The RC2 block cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
SECURITY_CIPHER_DES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher, using 56-bit

	keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher (also known as the Rijndael cipher) is a fixed block size cipher which use a key size of 128, 192 or 256 bits. This cipher is supported on Windows XP SP3 and later versions of the operating system.
SECURITY_CIPHER_SKIPJACK	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
SECURITY_CIPHER_BLOWFISH	The Blowfish block cipher was selected. This is a variable key length cipher up to 448 bits, using a 64-bit block size.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected. One of the following values will be returned:

Constant	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected. One of the following values will be returned:

--	--

Constant	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography. This key exchange algorithm is only supported on Windows XP SP3 and later versions of the operating system.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server. This member only has meaning for connections using the SSL or TLS protocols. One of the following values will be returned:

Constant	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertIssuer

A pointer to a string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertSubject

A pointer to a string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszFingerprint

A pointer to a string which contains a sequence of hexadecimal values that uniquely identify the server. This member is only used when a connection has been established using the Secure Shell (SSH) protocol.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Unicode: Implemented as Unicode and ANSI versions.

SMTPMESSAGE Structure

This structure provides information about the contents of a message and is used by the [SmtSubmitMessage](#) function.

```
typedef struct _SMTPMESSAGE
{
    LPCTSTR    lpszFrom;
    LPCTSTR    lpszTo;
    LPCTSTR    lpszCc;
    LPCTSTR    lpszBcc;
    LPCTSTR    lpszSubject;
    LPCTSTR    lpszText;
    LPCTSTR    lpszHTML;
    LPCTSTR    lpszAttach;
    UINT       nCharSet;
    UINT       nEncType;
    DWORD      dwReserved;
} SMTPMESSAGE, *LPSMTPMESSAGE;
```

Members

lpszFrom

A pointer to a string that specifies the email address of the person sending the message. This structure member must point to a valid address and cannot be NULL.

lpszTo

A pointer to a string that specifies the email addresses of one or more recipients. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member must point to at least one valid address and cannot be NULL.

lpszCc

A pointer to a string that specifies the email addresses of one or more recipients that will receive copies of the message. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member may be NULL or point to an empty string.

lpszBcc

A pointer to a string that specifies the email addresses of one or more recipients that will receive blind copies of the message. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member may be NULL or point to an empty string. Unlike the recipients specified by the *lpszTo* and *lpszCc* members, any addresses specified by this member will not be included in the header of the email message.

lpszSubject

A pointer to a string that specifies the subject of the message. This structure member may be NULL, in which case no subject will be included in the message.

lpszText

A pointer to a string which contains the body of the message as plain text. Each line of text contained in the string should be terminated with a carriage-return and linefeed (CRLF) pair, which is recognized as the end-of-line. If this structure member is NULL or points to an empty string, then the *lpszHTML* member must specify the body of the message.

lpszHTML

A pointer to a string which contains the message using HTML formatting. If the *lpszText* member is not NULL, then a multipart message will be created with both plain text and HTML

text as the alternative. This allows mail clients to select which message body they wish to display. If the *lpszText* member is NULL or points to an empty string, then the message will only contain HTML. Although this is supported, it is not recommended because older mail clients may be unable to display the message correctly.

lpszAttach

A pointer to a string which specifies one or more file attachments for the message. If multiple files are to be attached to the message, each file name must be separated by a semi-colon. It is recommended that you provide the complete path to the file. If this structure member is NULL or points to an empty string, the message will be created without attachments.

nCharSet

A integer value which specifies the character set to use when composing the message. A value of zero specifies that the default USASCII character set should be used. The following values may also be used:

Constant	Description
MIME_CHARSET_USASCII	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the first bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_CHARSET_ISO8859_1	An 8-bit character set for most western European languages such as English, French, Spanish and German. This character set is also commonly referred to as Latin1.
MIME_CHARSET_ISO8859_2	An 8-bit character set for most central and eastern European languages such as Czech, Hungarian, Polish and Romanian. This character set is also commonly referred to as Latin2.
MIME_CHARSET_ISO8859_5	An 8-bit character set for Cyrillic languages such as Russian, Bulgarian and Serbian.
MIME_CHARSET_ISO8859_6	An 8-bit character set for Arabic languages. Note that the application is responsible for displaying text that uses this character set. In particular, any display engine needs to be able to handle the reverse writing direction and analyze the context of the message to correctly combine the glyphs.
MIME_CHARSET_ISO8859_7	An 8-bit character set for the Greek language.
MIME_CHARSET_ISO8859_8	An 8-bit character set for the Hebrew language. Note that similar to Arabic, Hebrew uses a reverse writing direction. An application which displays this character should be capable of processing bi-directional text where a single message may include both right-to-left and left-to-right languages, such as Hebrew and English.
MIME_CHARSET_ISO8859_9	An 8-bit character set for the Turkish language. This character set is also commonly referred to as Latin5.

nEncType

A numeric identifier which specifies the encoding type to use when composing the message. A value of zero specifies that default 7bit encoding should be used. The following values may also be used:

Constant	Description
MIME_ENCODING_7BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the most significant bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_ENCODING_8BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long and all bits are used. 8-bit encoding is typically used with multibyte character sets and is the default encoding used with Unicode text.
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.

dwReserved

An unsigned integer value reserved for future use. This structure member should always have a value of zero.

Remarks

This structure is used to define the contents of a message that will be submitted for delivery using the **SmtplibSubmitMessage** function. It is required that you specify a sender, at least one recipient and a message body. All other structure members may be NULL or have a value of zero to indicate that either the value is not required, or that a default should be used. It is recommended that you initialize all of the structure members to a value of zero using the **ZeroMemory** function prior to populating the structure.

email addresses may be specified as simple addresses, or as commented addresses that include the sender's name or other information. For example, any one of these address formats are acceptable:

```
user@domain.tld
User Name <user@domain.tld>
user@domain.tld (User Name)
```

To specify multiple addresses, you should separate each address by a comma or semi-colon. Note that the *lpszFrom* member cannot specify multiple addresses, however it is permitted with the *lpszTo*, *lpszCc* and *lpszBcc* structure members. Each message must have at least one valid recipient, or the message cannot be submitted for delivery.

To send a message that contains HTML, it is recommended that you provide both a plain text version of the message body and an HTML formatted version. While it is permitted to send a

message that only contains HTML, some older mail clients may not be capable of displaying the message correctly. In some cases, anti-spam software will increase the spam score of messages that do not contain a plain text message body. This can result in your message being rejected or quarantined by the mail server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SMTPSERVER](#)

SMTPMESSAGEEX Structure

This structure provides information about the contents of a message and is used by the [SmtplibSubmitMessageEx](#) function.

```
typedef struct _SMTPMESSAGEEX
{
    DWORD    dwSize
    LPCTSTR  lpszFrom;
    LPCTSTR  lpszTo;
    LPCTSTR  lpszCc;
    LPCTSTR  lpszBcc;
    LPCTSTR  lpszDate;
    LPCTSTR  lpszSubject;
    LPCTSTR  lpszHeaders;
    LPCTSTR  lpszText;
    LPCTSTR  lpszHTML;
    LPCTSTR  lpszAttach;
    UINT     nCharSet;
    UINT     nEncType;
    DWORD    dwReserved;
} SMTPMESSAGEEX, *LPSMTPMESSAGEEX;
```

Members

dwSize

An integer value that specifies the size of the **SMTPMESSAGEEX** data structure. This must always be explicitly defined, and if the value is incorrect, an error will be returned. This structure member is used to ensure that the correct version of the structure is being passed to the function.

lpszFrom

A pointer to a string that specifies the email address of the person sending the message. This structure member must point to a valid address and cannot be NULL.

lpszTo

A pointer to a string that specifies the email addresses of one or more recipients. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member must point to at least one valid address and cannot be NULL.

lpszCc

A pointer to a string that specifies the email addresses of one or more recipients that will receive copies of the message. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member may be NULL or point to an empty string.

lpszBcc

A pointer to a string that specifies the email addresses of one or more recipients that will receive blind copies of the message. If multiple addresses are provided, they must be separated by commas or semi-colons. This structure member may be NULL or point to an empty string. Unlike the recipients specified by the *lpszTo* and *lpszCc* members, any addresses specified by this member will not be included in the header of the email message.

lpszDate

A pointer to a string that specifies the date and time for the message. This structure member may be NULL or point to an empty string. If the date is not specified, then the current date and time will be used by default. If a date is specified, it should be in the standard format as defined

by RFC822.

lpzSubject

A pointer to a string that specifies the subject of the message. This structure member may be NULL, in which case no subject will be included in the message.

lpzHeaders

A pointer to a string that specifies additional headers that should be included in the message. Header names should be separated from values by a colon, and multiple headers may be defined by separating them with a newline character. This structure member may be NULL, in which case no additional headers will be included in the message.

lpzText

A pointer to a string which contains the body of the message as plain text. Each line of text contained in the string should be terminated with a carriage-return and linefeed (CRLF) pair, which is recognized as the end-of-line. If this structure member is NULL or points to an empty string, then the *lpzHTML* member must specify the body of the message.

lpzHTML

A pointer to a string which contains the message using HTML formatting. If the *lpzText* member is not NULL, then a multipart message will be created with both plain text and HTML text as the alternative. This allows mail clients to select which message body they wish to display. If the *lpzText* member is NULL or points to an empty string, then the message will only contain HTML. Although this is supported, it is not recommended because older mail clients may be unable to display the message correctly.

lpzAttach

A pointer to a string which specifies one or more file attachments for the message. If multiple files are to be attached to the message, each file name must be separated by a semi-colon. It is recommended that you provide the complete path to the file. If this structure member is NULL or points to an empty string, the message will be created without attachments.

nCharSet

An integer value which specifies the character set to use when composing the message. A value of zero specifies that the default USASCII character set should be used. The following values may also be used:

Constant	Description
MIME_CHARSET_USASCII	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the first bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_CHARSET_ISO8859_1	An 8-bit character set for most western European languages such as English, French, Spanish and German. This character set is also commonly referred to as Latin1.
MIME_CHARSET_ISO8859_2	An 8-bit character set for most central and eastern European languages such as Czech, Hungarian, Polish and Romanian. This character set is also commonly referred to as Latin2.
MIME_CHARSET_ISO8859_5	An 8-bit character set for Cyrillic languages such as Russian, Bulgarian and Serbian.

MIME_CHARSET_ISO8859_6	An 8-bit character set for Arabic languages. Note that the application is responsible for displaying text that uses this character set. In particular, any display engine needs to be able to handle the reverse writing direction and analyze the context of the message to correctly combine the glyphs.
MIME_CHARSET_ISO8859_7	An 8-bit character set for the Greek language.
MIME_CHARSET_ISO8859_8	An 8-bit character set for the Hebrew language. Note that similar to Arabic, Hebrew uses a reverse writing direction. An application which displays this character should be capable of processing bi-directional text where a single message may include both right-to-left and left-to-right languages, such as Hebrew and English.
MIME_CHARSET_ISO8859_9	An 8-bit character set for the Turkish language. This character set is also commonly referred to as Latin5.

nEncType

A numeric identifier which specifies the encoding type to use when composing the message. A value of zero specifies that default 7bit encoding should be used. The following values may also be used:

Constant	Description
MIME_ENCODING_7BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the most significant bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
MIME_ENCODING_8BIT	Each character is encoded in one or more bytes, with each byte being 8 bits long and all bits are used. 8-bit encoding is typically used with multibyte character sets and is the default encoding used with Unicode text.
MIME_ENCODING_QUOTED	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.

dwReserved

An unsigned integer value reserved for future use. This structure member should always have a value of zero.

Remarks

This structure is used to define the contents of a message that will be submitted for delivery using

the **SmtplibSubmitMessageEx** function. It is required that you specify a sender, at least one recipient and a message body. Other structure members may be NULL or have a value of zero to indicate that either the value is not required, or that a default should be used. It is recommended that you initialize all of the structure members to a value of zero using the **ZeroMemory** function prior to populating the structure.

Note that you must explicitly define the size of the structure by setting the value of the **dwSize** member variable. This ensures that the correct version of the structure is being passed to the function. This structure is not compatible with the **SmtplibSubmitMessage** function and must only be used with **SmtplibSubmitMessageEx**.

Email addresses may be specified as simple addresses, or as commented addresses that include the sender's name or other information. For example, any one of these address formats are acceptable:

```
user@domain.tld
User Name <user@domain.tld>
user@domain.tld (User Name)
```

To specify multiple addresses, you should separate each address by a comma or semi-colon. Note that the **lpszFrom** member cannot specify multiple addresses, however it is permitted with the **lpszTo**, **lpszCc** and **lpszBcc** structure members. Each message must have at least one valid recipient, or the message cannot be submitted for delivery.

If you specify a message date by assigning a value to the **lpszDate** member, and it does not include any timezone information, Coordinated Universal Time (UTC) will be used by default. This is an important consideration if you provide input from a user, because in most cases they will not include the timezone and will assume the date and time they enter is for their current timezone.

If you wish to include additional headers in the message, you can specify them in a string. Each header consists of a name and value, separated by a colon (":") character. If you wish to define multiple headers, then you can separate them with a newline (e.g.: a linefeed character or combination of a carriage-return and linefeed). Extraneous leading and trailing whitespace are trimmed from header names and values. Invalid names or values will be ignored and will not generate an error.

To send a message that contains HTML, it is recommended that you provide both a plain text version of the message body and an HTML formatted version. While it is permitted to send a message that only contains HTML, some older mail clients may not be capable of displaying the message correctly. In some cases, anti-spam software will increase the spam score of messages that do not contain a plain text message body. This can result in your message being rejected or quarantined by the mail server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SMTPSERVER](#)

SMTPSERVER Structure

This structure provides connection information for a mail server and is used by the [SmtpSubmitMessage](#) function.

```
typedef struct _SMTPSERVER
{
    LPCTSTR lpszHostName;
    LPCTSTR lpszUserName;
    LPCTSTR lpszPassword;
    UINT nHostPort;
    UINT nTimeout;
    DWORD dwOptions;
    DWORD dwReserved;
} SMTPSERVER, *LPSMTPSERVER;
```

Members

lpszHostName

A pointer to a string that specifies the host name or IP address of the mail server. This structure member cannot be NULL.

lpszUserName

A pointer to a string that specifies the username that will be used to authenticate the client session. If the mail server does not require authentication, this structure member can be NULL or point to an empty string.

lpszPassword

A pointer to a string that specifies the password that will be used to authenticate the client session. If the mail server does not require authentication, this structure member can be NULL or point to an empty string.

nHostPort

An integer value that specifies the port number used to establish the connection. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 25. An alternative port is 587, which is commonly used by authenticated clients to submit messages for delivery. For implicit SSL connections, the default port number is 465.

nTimeout

An integer value that specifies the number of seconds that the client will wait for a response from the server before failing the operation. A value of zero specifies the default timeout period of 20 seconds.

dwOptions

An unsigned integer that specifies one or more options. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Constant	Description
SMTP_OPTION_NONE	No additional options are specified when establishing a connection with the server. A standard, non-secure connection will be used.
SMTP_OPTION_EXTENDED	Extended SMTP commands should be used if possible. This option enables features such as

	authentication and delivery status notification. If this option is not specified, the library will not attempt to use any extended features. This option is automatically enabled if a username and password are specified, or if the connection is established on port 587, because submitting messages for delivery using this port typically requires client authentication.
SMTP_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
SMTP_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
SMTP_OPTION_SECURE	This option specifies that a secure connection should be established with the server and requires that the server support either the SSL or TLS protocol. The client will initiate the secure session using the STARTTLS command.
SMTP_OPTION_SECURE_IMPLICIT	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using either the SSL or TLS protocol, and the secure session must be negotiated immediately after the connection has been established.

dwReserved

An unsigned integer value reserved for future use. This structure member should always have a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SMTPMESSAGE](#)

SMTPTTRANSFERSTATUS Structure

This structure is used by the [SmtpGetTransferStatus](#) function to return information about a message being submitted for delivery.

```
typedef struct _SMTPTTRANSFERSTATUS
{
    DWORD    dwBytesTotal;
    DWORD    dwBytesCopied;
    DWORD    dwBytesPerSecond;
    DWORD    dwTimeElapsed;
    DWORD    dwTimeEstimated;
} SMTPTTRANSFERSTATUS, *LPSMTPTTRANSFERSTATUS;
```

Members

dwBytesTotal

The total number of bytes that will be transferred. If the size of the message cannot be determined, this value will be zero.

dwBytesCopied

The total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the transfer is completed. This is based on the average number of bytes transferred per second.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

SYSTEMTIME Structure

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include windows.h.

SocketWrench Library

A general purpose TCP/IP networking library for developing client and server applications.

Reference

- [Functions](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

File Name	CSWSKV10.DLL
Version	10.0.1468.2518
LibID	1437629B-0693-44DE-93ED-1482DBEFE8DC
Import Library	CSWSKV10.LIB
Dependencies	None
Standards	RFC 768, RFC 791, RFC 793

Overview

At the core of all of the SocketTools networking libraries is the Windows Sockets API. This provides a low level interface for sending and receiving data over the Internet or a local intranet using the Transmission Control Protocol (TCP) and/or User Datagram Protocol (UDP). The SocketWrench library provides a simpler interface to the Windows Sockets API, without sacrificing features or functionality. Using SocketWrench, you can easily create client and server applications while avoiding many of the mundane tasks and common problems that developers face when building Internet applications.

This library supports secure connections using the TLS 1.2 protocol and can also be used to create secure, customized server applications. Both implicit and explicit SSL connections are supported, enabling the library to work with a wide variety of client and server applications without requiring that you use third-party libraries or Microsoft's CryptoAPI.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This library provides an implementation of a multithreaded server which should only be used with languages that support the creation of multithreaded applications. It is important that you do not link against static libraries which were not built with support for threading.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-

bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

SocketWrench Library Functions

Function	Description
InetAbort	Abort the connection and immediately close the socket
InetAccept	Accept a connection request from a remote host
InetAcceptEx	Accept a client connection on a listening socket with additional options
InetAsyncAccept	Accept an asynchronous connection request from a remote host
InetAsyncAcceptEx	Accept a non-blocking connection on a listening socket
InetAsyncConnect	Connect asynchronously to the specified server
InetAsyncConnectEx	Connect asynchronously to the specified server
InetAsyncListen	Listen for client connections on an asynchronous socket
InetAsyncListenEx	Listen for client connections on an asynchronous socket with additional options
InetAttachSocket	Attach the socket handle to the specified process
InetAttachThread	Attach the specified socket to another thread
InetCancel	Cancel a blocking operation
InetClientBroadcast	Write data to all other clients that are connected to the same server
InetCompareAddress	Compare two IP addresses to determine if they are identical
InetConnect	Connect to the specified server
InetConnectEx	Connect to the specified server
InetCreateSecurityCredentials	Create a new security credentials structure
InetDeleteSecurityCredentials	Delete a previously created security credentials structure
InetDetachSocket	Detach the socket handle from the current process
InetDisableEvents	Disable asynchronous event notification
InetDisableSecurity	Disable secure communication with the remote host
InetDisableTrace	Disable logging of socket function calls to the trace log
InetDisconnect	Disconnect from the current server
InetEnableEvents	Enable asynchronous event notification
InetEnableSecurity	Enable secure communication with the remote host
InetEnableTrace	Enable logging of socket function calls to a file
InetEnumHostAliases	Return a byte array that contains all the aliases for a specified host
InetEnumNetworkAddresses	Return the list of network addresses that are configured for the local host
InetEnumServerClients	Returns a list of active client sessions established with the specified server
InetEnumServerClientsByAddress	Returns a list of active client sessions that match the specified IP address
InetEventProc	Callback method that processes events generated on the socket
InetFindClientMoniker	Returns a handle to the client socket which matches the specified moniker
InetFlush	Flush the send and receive buffers
InetFormatAddress	Convert an IP address in binary format into a printable string
InetFreezeEvents	Suspend or resume event handling by the application
InetGetAdapterAddress	Return the IP or MAC assigned to the specified network adapter
InetGetAddress	Convert an IP address string to a binary format
InetGetAddressFamily	Return the address family for the specified IP address

InetGetBlockingSocket	Return the handle for the socket which is blocked in the current thread
InetGetDefaultHostFile	Return the fully qualified path name of the host file on the local system
InetGetClientData	Returns the application defined data associated with the specified client session
InetGetClientHandle	Returns the handle for a specific client session based on its ID number
InetGetClientId	Returns the unique ID number assigned to the specified client session
InetGetClientIdleTime	Returns the amount of time the specified client session has been idle
InetGetClientMoniker	Returns the string alias associated with the specified client session
InetGetClientPriority	Returns the current priority for the specified client session
InetGetClientServer	Returns a socket handle to the server for the specified client socket
InetGetClientServerById	Returns a socket handle to the server for the specified session identifier
InetGetClientThreadId	Returns the thread ID for the specified client session
InetGetClientThreads	Returns the number of client session threads created by the server
InetGetErrorString	Return a description for the specified error code
InetGetExternalAddress	Return the external IP address assigned to the local system
InetGetHostAddress	Return the IP address assigned to the specified hostname
InetGetHostFile	Return the name of the host file
InetGetHostName	Return the hostname assigned to the specified IP address
InetGetLastError	Return the last error code
InetGetLocalAddress	Return the local IP address and port number for a socket
InetGetLocalName	Return the hostname assigned to the local system
InetGetLockedServer	Return the handle to the server which has been locked
InetGetOption	Return the current socket options
InetGetPeerAddress	Return the IP address of the peer that the socket is connected to
InetGetPhysicalAddress	Return the media access control (MAC) address for the primary network adapter
InetGetSecurityInformation	Return information about the security characteristics of a connection
InetGetServerClient	Return the handle for the last client connection accepted by the server
InetGetServerData	Returns the application defined data associated with the specified server
InetGetServerPriority	Return the current priority assigned to the specified server
InetGetServerStackSize	Return the initial size of the stack allocated for threads created by the server
InetGetServerStatus	Returns the status of the specified server
InetGetServerThreadId	Returns the thread ID for the specified server
InetGetServiceName	Return the service name associated with a specified port number
InetGetServicePort	Return the port number associated with a service name
InetGetStatus	Report what sort of socket operation is in progress
InetGetStreamInfo	Return information about the current stream read or write operation
InetGetThreadClient	Return the handle for the client session that is being managed by the specified thread
InetGetTimeout	Return the timeout interval for blocking operations, in seconds
InetHostNameToUnicode	Converts the canonical form of a host name to its Unicode version
InetInitialize	Initialize the library and validate the specified user license key at runtime
InetIsAddressNull	Determine if the specified IP address is a null address
InetIsAddressRoutable	Determine if the specified IP address is routable over the Internet

InetIsBlocking	Determine if the socket is performing a blocking operation
InetIsClosed	Determine if the remote host has closed its socket
InetIsConnected	Determine if the socket is connected to a remote host
InetIsListening	Determine if the socket is listening for a connection
InetIsProtocolAvailable	Determine if the specified protocol and address family are supported
InetIsReadable	Determine if data can be read from the remote process
InetIsUrgent	Determine if there is any out-of-band (OOB) data available to be read
InetIsWritable	Determine if data can be written to the remote process
InetListen	Listen for client connections on the specified socket
InetListenEx	Listen for client connections on the specified socket with additional options
InetMatchHostName	Match a host name against a list of addresses including wildcards
InetNormalizeHostName	Return the canonical form of a host name
InetPeek	Read data from the socket without removing it from the socket buffer
InetRead	Read data from the socket
InetReadEx	Read data from the socket, with extended functionality
InetReadLine	Read a line of data from the socket, storing it in a string buffer
InetReadStream	Read a stream of data from the socket
InetRegisterEvent	Register an event callback function
InetReject	Reject a pending client connection request
InetServerAsyncNotify	Enable or disable asynchronous notification of changes in server status
InetServerBroadcast	Write data to all active clients currently connected to the specified server
InetServerLock	Lock the specified server, causing all other client threads to block until it is unlocked
InetServerRestart	Restart the server, terminating all active client sessions
InetServerResume	Resume accepting client connections on the specified server
InetServerStart	Begin listening for client connections on the specified address and port
InetServerStop	Stop listening for connections and terminate all client sessions
InetServerStopEx	Stop listening for connections and wait for the server to terminate
InetServerSuspend	Suspend accepting client connections on the specified server
InetServerSuspendEx	Suspend accepting client connections and optionally reject or disconnect clients
InetServerThrottle	Limit the number of active client connections, connections per address and connection rate
InetServerUnlock	Unlock the specified server, allowing other client threads to resume execution
InetSetClientData	Associate application defined data with the specified client session
InetSetClientMoniker	Associate a unique string alias with the specified client session
InetSetClientPriority	Set the priority for the specified client session
InetSetServerData	Associate application defined data with the specified server
InetSetServerPriority	Change the priority assigned to the specified server
InetSetServerStackSize	Change the initial size of the stack allocated for threads created by the server
InetSetHostFile	Specify the name of an alternate host table
InetSetLastError	Set the last error code
InetSetOption	Set one or more options for the current socket
InetSetTimeout	Set the interval used when waiting for a blocking operation to complete

InetShutdown	Disable reception or transmission of data
InetStoreStream	Read a stream of data from the remote host and store it in a file
InetUninitialize	Terminate use of the library by the application
InetWrite	Write data to the socket
InetWriteEx	Write data to the socket, with extended functionality
InetWriteLine	Write a line of data to the socket, terminated with a carriage-return and linefeed
InetWriteStream	Write a stream of data to the socket

InetAbort Function

```
INT WINAPI InetAbort(  
    SOCKET hSocket  
);
```

Immediately close the socket without waiting for any remaining data to be written out.

Parameters

hSocket

Handle to the socket.

Return Value

If the function succeeds, the return value is 0. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

The **InetAbort** function should only be used when the connection must be closed immediately before the application terminates. This function should only be used to abort client connections and should not be used with passive (listening) sockets. Server applications that need to abort an incoming client connection should use the **InetReject** function.

In most cases, the application should call the **InetDisconnect** function to gracefully close the connection to the remote host. Aborting the connection will discard any buffered data and may cause errors or result in unpredictable behavior.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

See Also

[InetCancel](#), [InetReject](#), [InetDisconnect](#)

InetAccept Function

```
SOCKET WINAPI InetAccept(  
    SOCKET hSocket,  
    UINT nTimeout  
);
```

The **InetAccept** function is used to accept a client connection on a listening socket.

This function is included for backwards compatibility with legacy applications. New projects should use the **InetServerStart** function to create a server application.

Parameters

hSocket

Handle to the listening socket.

nTimeout

The number of seconds that the server will wait for the connection to complete before failing the operation.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

To prevent this function from blocking the main user interface thread, the application should create a background worker thread and accept the connection by calling **InetAccept** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

When a connection is accepted by the server, the original listening socket continues to listen for more connections. The socket handle returned by **InetAccept** should be used to exchange information with the client.

To enable asynchronous event notification, use the **InetEnableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetConnect](#), [InetEnableEvents](#), [InetListen](#), [InetReject](#), [InetServerStart](#)

InetAcceptEx Function

```
SOCKET WINAPI InetAcceptEx(  
    SOCKET hSocket,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSECURITYCREDENTIALS LpCredentials  
);
```

The **InetAcceptEx** function is used to accept a client connection on a listening socket, and to set certain attributes of that connection.

This function is included for backwards compatibility with legacy applications. New projects should use the **InetServerStart** function to create a server application.

Parameters

hSocket

Handle to the socket.

nTimeout

The number of seconds that the server will wait for a client connection before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.
INET_OPTION_INLINE	This option controls how urgent (out-of-band) data is handled when reading data from the socket. If set, urgent data is placed in the data stream along with non-urgent data.
INET_OPTION_SECURE	This option determines if a secure connection is established with the remote host.
INET_OPTION_SECURE_FALLBACK	This option specifies the server should permit the use of less secure cipher suites for compatibility with legacy clients. If this option is specified, the server will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_FREETHREAD	This option specifies the socket returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access

to the socket is synchronized across multiple threads.

lpCredentials

Pointer to credentials structure [SECURITYCREDENTIALS](#). This may be NULL, unless *dwOptions* includes INET_OPTION_SECURE. When INET_OPTION_SECURE is used, the fields *dwSize*, *lpzCertStore*, and *lpzCertName* must be defined, while other fields may be left undefined. Set *dwSize* to the size of the [SECURITYCREDENTIALS](#) structure.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

To prevent this function from blocking the main user interface thread, the application should create a background worker thread and accept the connection by calling **InetAcceptEx** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

When a connection is accepted by the server, the original listening socket continues to listen for more connections. The socket handle returned by **InetAcceptEx** should be used to exchange information with the client.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **InetAttachThread** function.

Specifying the INET_OPTION_FREETHREAD option enables any thread to call any function using the socket handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the socket is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same socket handle.

To enable asynchronous event notification, use the **InetEnableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetAccept](#), [InetConnect](#), [InetListen](#), [InetConnectEx](#), [InetListenEx](#), [InetServerStart](#)

InetAsyncAccept Function

```
SOCKET WINAPI InetAsyncAccept(  
    SOCKET hSocket,  
    UINT nTimeout,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **InetAsyncAccept** function is used to accept a client connection on a listening socket.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

New projects should use the **InetServerStart** function to create a server application.

Parameters

hSocket

Handle to the listening socket.

nTimeout

The number of seconds that the server will wait for the connection to complete before failing the operation. This value is used only if *hWnd* is NULL.

hEventWnd

The handle to the event notification window. This window receives messages which notify the application of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

When a connection is accepted by the server, the original listening socket continues to listen for more connections. The socket handle returned by **InetAsyncAccept** should be used to exchange information with the client.

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the socket handle. One or more of the following event identifiers may be sent:

Constant	Description
INET_EVENT_DISCONNECT	The remote host has closed the connection. The process should read any remaining data and disconnect.
INET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the process has read at least some

	of the data from the socket. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_WRITE	The process can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the socket is in asynchronous mode.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetAccept](#), [InetConnect](#), [InetListen](#), [InetReject](#), [InetServerStart](#)

InetAsyncAcceptEx Function

```
SOCKET WINAPI InetAsyncAcceptEx(  
    SOCKET hSocket,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSECURITYCREDENTIALS lpCredentials  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **InetAsyncAcceptEx** function is used to accept a client connection on a listening socket and set certain attributes of that connection.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

New projects should use the **InetServerStart** function to create a server application.

Parameters

hSocket

Handle to the listening socket.

nTimeout

The number of seconds that the server will wait for a client connection before failing the operation. This value is only used with blocking connections.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option is commonly used by server applications.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.
INET_OPTION_INLINE	This option controls how urgent (out-of-band) data is handled when reading data from the socket. If set, urgent data is placed in the data stream along with non-urgent data.
INET_OPTION_SECURE	This option determines if a secure connection is established with the remote host.

INET_OPTION_SECURE_FALLBACK	This option specifies the server should permit the use of less secure cipher suites for compatibility with legacy clients. If this option is specified, the server will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_FREETHREAD	This option specifies the socket returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the socket is synchronized across multiple threads.

lpCredentials

Pointer to credentials structure [SECURITYCREDENTIALS](#). This may be NULL, unless dwOptions includes INET_OPTION_SECURE. When INET_OPTION_SECURE is used, the fields *dwSize*, *lpzCertStore*, and *lpzCertName* must be defined, while other fields may be left undefined. Set *dwSize* to the size of the **SECURITYCREDENTIALS** structure.

hEventWnd

The handle to the event notification window. This window receives messages which notify the application of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

When a connection is accepted by the server, the original listening socket continues to listen for more connections. The socket handle returned by **InetAsyncAccept** or **InetAsyncAcceptEx** should be used to exchange information with the client.

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the socket handle. One or more of the following event identifiers may be sent:

Constant	Description
INET_EVENT_DISCONNECT	The remote host has closed the connection. The process should read any remaining data and disconnect.
INET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the process has read at least some of the data from the socket. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_WRITE	The process can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the socket is in

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **InetAttachThread** function.

Specifying the INET_OPTION_FREETHREAD option enables any thread to call any function using the socket handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the socket is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same socket handle.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetAccept](#), [InetConnect](#), [InetListen](#), [InetAcceptEx](#), [InetConnectEx](#), [InetListenEx](#), [InetServerStart](#)

InetAsyncConnect Function

```
SOCKET WINAPI InetAsyncConnect(  
    LPCTSTR lpszHostName,  
    UINT nPort,  
    UINT nProtocol,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSECURITYCREDENTIALS lpCredentials,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **InetAsyncConnect** function is used to establish a connection with a server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **InetConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

Parameters

lpszHostName

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nPort

The port number the server is listening on; a value of zero specifies that the default port number should be used.

nProtocol

The protocol to be used when establishing the connection. This may be one of the following values:

Constant	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This is the default protocol.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. Note that UDP is unreliable in that there is no way for the sender to know that the receiver has actually received the datagram.

nTimeout

The number of seconds to wait for a response before failing the current operation.

dwOptions

An unsigned integer used to specify one or more socket options. This parameter is constructed by using the bitwise Or operator with any of the following values:

--	--

Constant	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_DONTROUTE	This option specifies default routing should not be used. This option should not be specified unless absolutely necessary.
INET_OPTION_KEEPAIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_NODELAY	This option disables the Nagle algorithm. By default, small amounts of data written to the socket are buffered, increasing efficiency and reducing network congestion. However, this buffering can negatively impact the responsiveness of certain applications. This option disables this buffering and immediately sends data packets as they are written to the socket.
INET_OPTION_RESERVEDPORT	This option specifies the socket should be bound to an unused port number less than 1024, which is typically reserved for well-known system services. If this option is specified, the process may require administrative privileges.
INET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
INET_OPTION_SECURE	This option specifies that a secure connection should be established with the remote host. The specific version of TLS and other security related options are provided in the <i>lpCredentials</i> parameter. If the <i>lpCredentials</i> parameter is NULL, the connection will default to using TLS 1.2 and the strongest cipher suites available. Older versions of Windows prior to Windows 7 and Windows Server 2008 R2 only support TLS 1.0 and secure connections will automatically downgrade on those platforms.
INET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_PREFER_IPV6	This option specifies the client should prefer the

	use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
INET_OPTION_FREETHREAD	This option specifies the socket returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the socket is synchronized across multiple threads.

lpCredentials

A pointer to a [SECURITYCREDENTIALS](#) structure. This parameter is only used if INET_OPTION_SECURE is specified for a TCP connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields *dwSize*, *lpszCertStore*, and *lpszCertName* must be defined, while other fields may be left undefined. Set *dwSize* to the size of the **SECURITYCREDENTIALS** structure.

hEventWnd

The handle to the event notification window. This window receives messages which notify the application of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

When this function is called with UDP as the specified protocol, it does not actually establish a connection. Instead, it simply establishes a default destination IP address and port that is used with subsequent **InetRead** and **InetWrite** calls.

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the socket handle. One or more of the following event identifiers may be sent:

Constant	Description
INET_EVENT_CONNECT	The connection to the remote host has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
INET_EVENT_DISCONNECT	The remote host has closed the connection. The process should read any remaining data and disconnect.
INET_EVENT_READ	Data is available to read by the calling process. No additional

	messages will be posted until the process has read at least some of the data from the socket. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_WRITE	The process can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the socket is in asynchronous mode.

To cancel asynchronous notification and return the socket to a blocking mode, use the **InetDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetAsyncConnectEx](#), [InetConnect](#), [InetDisableEvents](#), [InetDisconnect](#), [InetEnableEvents](#), [InetInitialize](#)

InetAsyncConnectEx Function

```
SOCKET WINAPI InetAsyncConnectEx(  
    LPCTSTR lpszHostName,  
    UINT nPort,  
    UINT nProtocol,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPCTSTR lpszLocalAddress,  
    UINT nLocalPort,  
    LPSECURITYCREDENTIALS LpCredentials,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **InetAsyncConnectEx** function is used to establish a connection with a server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **InetConnectEx** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

lpszHostName

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nPort

The port number the server is listening on; a value of zero specifies that the default port number should be used.

nProtocol

The protocol to be used when establishing the connection. This may be one of the following values:

Constant	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This is the default protocol.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. Note that UDP is unreliable in that there is no way for the sender to know that the receiver has actually received the datagram.

nTimeout

The number of seconds to wait for a response before failing the current operation.

dwOptions

An unsigned integer used to specify one or more socket options. The following values are

recognized:

Constant	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_DONTROUTE	This option specifies default routing should not be used. This option should not be specified unless absolutely necessary.
INET_OPTION_KEEPAIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_NODELAY	This option disables the Nagle algorithm. By default, small amounts of data written to the socket are buffered, increasing efficiency and reducing network congestion. However, this buffering can negatively impact the responsiveness of certain applications. This option disables this buffering and immediately sends data packets as they are written to the socket.
INET_OPTION_RESERVEDPORT	This option specifies the socket should be bound to an unused port number less than 1024, which is typically reserved for well-known system services. If this option is specified, the process may require administrative privileges.
INET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
INET_OPTION_SECURE	This option specifies that a secure connection should be established with the remote host. The specific version of TLS and other security related options are provided in the <i>lpCredentials</i> parameter. If the <i>lpCredentials</i> parameter is NULL, the connection will default to using TLS 1.2 and the strongest cipher suites available. Older versions of Windows prior to Windows 7 and Windows Server 2008 R2 only support TLS 1.0 and secure connections will automatically downgrade on those platforms.
INET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.

INET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
INET_OPTION_FREETHREAD	This option specifies the socket returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the socket is synchronized across multiple threads.

lpszLocalAddress

A pointer to a string that specifies the local IP address that the socket should be bound to. If this parameter is NULL, then an appropriate address will automatically be used. A specific address should only be used if it is required by the application.

nLocalPort

The local port number that the socket should be bound to. If this parameter is set to zero, then an appropriate port number will automatically be used. A specific port number should only be used if it is required by the application.

lpCredentials

A pointer to a [SECURITYCREDENTIALS](#) structure. This parameter is only used if INET_OPTION_SECURE is specified for a TCP connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields *dwSize*, *lpszCertStore*, and *lpszCertName* must be defined, while other fields may be left undefined. Set *dwSize* to the size of the **SECURITYCREDENTIALS** structure.

hEventWnd

The handle to the event notification window. This window receives messages which notify the application of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

It is not recommend that you disable the Nagle algorithm by specifying the INET_OPTION_NODELAY flag unless it is absolutely required. Doing so can have a significant, negative impact on the performance of the application and network.

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if

an error has occurred. The *wParam* parameter contains the socket handle. One or more of the following event identifiers may be sent:

Constant	Description
INET_EVENT_CONNECT	The connection to the remote host has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
INET_EVENT_DISCONNECT	The remote host has closed the connection. The process should read any remaining data and disconnect.
INET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the process has read at least some of the data from the socket. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_WRITE	The process can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the socket is in asynchronous mode.

To cancel asynchronous notification and return the socket to a blocking mode, use the **InetDisableEvents** function.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **InetAttachThread** function.

Specifying the INET_OPTION_FREETHREAD option enables any thread to call any function using the socket handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the socket is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same socket handle.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetAsyncConnect](#), [InetConnect](#), [InetDisableEvents](#), [InetDisconnect](#), [InetEnableEvents](#), [InetInitialize](#)

InetAsyncListen Function

```
SOCKET WINAPI InetAsyncListen(  
    LPCTSTR lpszLocalAddress,  
    UINT nLocalPort,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **InetAsyncListen** function creates a listening socket.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

New projects should use the **InetServerStart** function to create a server application.

Parameters

lpszLocalAddress

A pointer to a string which specifies the local IP address that the socket should be bound to. If this parameter is NULL or points to an empty string, a client may establish a connection using any valid network interface configured on the local system. If an address is specified, then a client may only establish a connection with the system using that address.

nLocalPort

The local port number that the socket should be bound to. This value must be greater than zero. Port numbers less than 1024 are considered reserved ports and may require that the process execute with administrative privileges and/or require changes to the default firewall rules to permit inbound connections.

hEventWnd

The handle to the event notification window. This window receives messages which notify the application of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

The socket option INET_OPTION_REUSEADDRESS is enabled by default when calling the **InetAsyncListen** function. This allows an application to re-use a local address and port number when creating the listening socket. If this behavior is not desired, use the **InetAsyncListenEx**

function instead.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the function will fail.

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the socket handle. One or more of the following event identifiers may be sent:

Constant	Description
INET_EVENT_ACCEPT	The process has received a connection request from a client and should accept the connection using the InetAsyncAccept function. This event is only generated for server applications which have created an asynchronous socket using the InetAsyncListen function.

To cancel asynchronous notification and return the socket to a blocking mode, use the **InetDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cswsock10.h

Import Library: cswskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetAccept](#), [InetDisableEvents](#), [InetEnableEvents](#), [InetInitialize](#), [InetListen](#), [InetReject](#), [InetServerStart](#)

InetAsyncListenEx Function

```
SOCKET WINAPI InetAsyncListenEx(  
    LPCTSTR lpszLocalAddress,  
    UINT nLocalPort,  
    UINT nBackLog,  
    DWORD dwOptions,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **InetAsyncListenEx** function creates a listening socket and specifies the maximum number of connection requests that will be queued.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

New projects should use the **InetServerStart** function to create a server application.

Parameters

lpszLocalAddress

A pointer to a string which specifies the local IP address that the socket should be bound to. If this parameter is NULL or points to an empty string, a client may establish a connection using any valid network interface configured on the local system. If an address is specified, then a client may only establish a connection with the system using that address.

nLocalPort

The local port number that the socket should be bound to. This value must be greater than zero. Port numbers less than 1024 are considered reserved ports and may require that the process execute with administrative privileges and/or require changes to the default firewall rules to permit inbound connections.

nBacklog

The maximum length of the queue allocated for pending client connections. A value of zero specifies that the size of the queue should be set to a maximum reasonable value. On Windows server platforms, the maximum value is large enough to queue several hundred pending connections.

dwOptions

An unsigned integer used to specify one or more socket options. The following values are supported:

Constant	Description
INET_OPTION_NONE	No option specified. If the address and port number are in use by another application or a closed socket which was listening on this port is still in the TIME_WAIT state, the function will fail.
INET_OPTION_REUSEADDRESS	This option enables a server application to listen for connections using the specified address and port number even if they were in use recently. This is typically used to enable an application to close the

	listening socket and immediately reopen it without getting an error that the address is in use.
INET_OPTION_EXCLUSIVE	This option specifies the local address and port number is for the exclusive use by the current process, preventing another application from forcibly binding to the same address. If another process has already bound a socket to the address provided by the caller, this function will fail.
INET_OPTION_RESERVEDPORT	This option specifies the listening socket should be bound to an unused port number less than 1024, which is typically reserved for well-known system services. If this option is specified, the process may require administrative privileges and firewall rules that will permit a client to establish a connection with the service.

hEventWnd

The handle to the event notification window. This window receives messages which notify the application of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

If the INET_OPTION_REUSEADDRESS option is not specified, an error may be returned if a listening socket was recently created for the same local address and port number. By default, once a listening socket is closed there is a period of time that all applications must wait before the address can be reused (this is called the TIME_WAIT state). The actual amount of time depends on the operating system and configuration parameters, but is typically two to four minutes. Specifying this option enables an application to immediately re-use a local address and port number that was previously in use.

If the INET_OPTION_EXCLUSIVE option is specified, the local address and port number cannot be used by another process until the listening socket is closed. This can prevent another application from forcibly binding to the same listening address as your server. This option can be useful in determining whether or not another process is already bound to the address you wish to use, but it may also prevent your server application from restarting immediately, regardless if the INET_OPTION_REUSEADDRESS option has also been specified.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed

and configured, otherwise the function will fail.

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the socket handle. One or more of the following event identifiers may be sent:

Constant	Description
INET_EVENT_ACCEPT	An incoming client connection is pending. The connection will be assigned to a new socket. This event is only generated if the socket is in asynchronous mode.

To cancel asynchronous notification and return the socket to a blocking mode, use the **InetDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cswsock10.h

Import Library: cswskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetAccept](#), [InetDisableEvents](#), [InetEnableEvents](#), [InetListenEx](#), [InetServerStart](#)

InetAttachSocket Function

```
SOCKET WINAPI InetAttachSocket(  
    SOCKET hSocket  
    DWORD dwProcessId  
);
```

The **InetAttachThread** function attaches the specified socket handle to another thread.

Parameters

hSocket

Handle to the socket.

dwProcessId

The process ID for the process that currently owns the socket handle. This value may be zero to specify the current process.

Return Value

If the function succeeds, the return value is the handle to the attached socket. If the function fails, the return value is `INVALID_SOCKET`. To get extended error information, call **InetGetLastError**.

Remarks

The **InetAttachSocket** function enables an application to attach an external socket handle to the current thread and initializes it for use by the library. An external socket is any socket handle created directly by the Windows Sockets API or by a third-party library. If the *dwProcessId* parameter is zero or specifies the current process ID, then the function checks to see if the socket was created by this library. If it was, then its owner context is switched to the current thread; if the socket was created externally, then it is initialized for use by the library and attached to the current thread.

If the *dwProcessId* parameter specifies another process, the socket will be duplicated into the current process, attached to the current thread and the original socket handle will be closed in the other process. This enables an application to effectively take control of a connection created by another process. The original socket handle must be inheritable by the by the current process and must be an actual Windows socket handle, not a pseudo-handle. This functionality is only supported on Windows NT 4.0 and later versions of the operating system with the Microsoft TCP/IP stack. Note that Layered Service Providers (LSPs) may interfere with the ability to inherit handles across processes.

The attached socket is initialized in a blocking state, even if was originally using asynchronous socket events. If the application requires that the socket use events, it must explicitly call **InetEnableEvents** using the handle returned by this function.

In most cases, the handle returned by this function will be the same value as the *hSocket* parameter, however an application should never make the assumption that this will be the case. If **InetAttachSocket** returns a socket handle that has a different value than the *hSocket* parameter, this indicates that the original handle has been destroyed and should never be used in subsequent function calls.

This function should never be used with a secure socket connection because the attached socket will not have the security context required to encrypt and decrypt the data exchanged with the remote host.

Example

To demonstrate how to pass sockets between processes, this example will use two programs; one acting as a server to listen for client connections and accept them, the other inheriting the client socket and echoing back anything the client sends. The first program will create the listening socket, and when a client connects, it will call **CreateProcess** to create a new child process to handle that connection.

```
SOCKET hServer;
SOCKET hClient;

// Initialize the library

if (!InetInitialize(CSTOOLS10_LICENSE_KEY, NULL))
    return;

// Listen for incoming client connections

if ((hServer = InetListen(NULL, nLocalPort)) == INVALID_SOCKET)
    return;

while (TRUE)
{
    hClient = InetAccept(hServer, 10);

    if (hClient == INVALID_SOCKET)
    {
        // If InetAccept has timed-out, then simply loop back and attempt
        // to continue accepting connections; otherwise, exit the loop

        if (InetGetLastError() != ST_ERROR_OPERATION_TIMEOUT)
            break;
    }
    else
    {
        STARTUPINFO si;
        PROCESS_INFORMATION pi;
        CHAR szCommandLine[512];
        BOOL bResult;

        // Detach the socket, which will free the memory that the library
        // has allocated for it without actually destroying the socket handle;
        // the child process will close the handle in this process when it
        // attaches to it
        InetDetachSocket(hClient, 0);

        // Initialize the STARTUPINFO structure
        ZeroMemory(&si, sizeof(si));

        // Create the command line arguments, passing the current
        // process ID and socket handle to the new process

        wsprintf(szCommandLine, "%s %lu %lu",
                lpzAppName,
                (DWORD)GetCurrentProcessId(),
                (DWORD)hClient);

        // Create the child process

        bResult = CreateProcess(NULL,
                                szCommandLine,
```

```

        NULL, NULL,
        TRUE,
        CREATE_DEFAULT_ERROR_MODE,
        NULL, NULL,
        &si, &pi);

    if (!bResult)
        InetDisconnect(hClient);
}
}

```

```

InetDisconnect(hServer);
InetUninitialize();

```

The second program attaches to the socket handle that was passed to it by the parent process. It goes into a loop, reading any data sent to it by the client and sending the same data back. When the client disconnects, the **InetRead** function will return 0, it will exit the loop and the process will terminate.

```

SOCKET hSocket;
SOCKET hClient;
DWORD dwProcessId;

// Initialize the library

if (!InetInitialize(CSTOOLS10_LICENSE_KEY, NULL))
    return;

// Process command line arguments that were passed to us
// by the server process

dwProcessId = (DWORD)atoi(argv[1]);
hClient = (SOCKET)atoi(argv[2]);

// Attach to the hClient socket that the server passed
// to us; this will close the socket in the server process

hSocket = InetAttachSocket(hClient, dwProcessId);

if (hSocket != INVALID_SOCKET)
{
    BYTE cBuffer[512];
    int nRead;

    do
    {
        // Read any data sent to us by the client
        nRead = InetRead(hSocket, cBuffer, sizeof(cBuffer));

        // Echo the data we have read back to the client
        if (nRead > 0)
            InetWrite(hSocket, cBuffer, nRead);
    }
    while (nRead > 0);

    InetDisconnect(hSocket);
}
}

```

```
InetUninitialize();
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cswsock10.h

Import Library: cswskv10.lib

See Also

[InetAttachThread](#), [InetDetachSocket](#), [InetInitialize](#)

InetAttachThread Function

```
DWORD WINAPI InetAttachThread(  
    SOCKET hSocket  
    DWORD dwThreadId  
);
```

The **InetAttachThread** function attaches the specified socket handle to another thread.

Parameters

hSocket

Handle to the socket.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the socket handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

When a socket handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in an application to create the socket, and then pass that handle to another worker thread. The **InetAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **InetAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **InetCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **InetUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetAccept](#), [InetCancel](#), [InetAsyncConnect](#), [InetConnect](#), [InetDisconnect](#), [InetUninitialize](#)

InetCancel Function

```
INT WINAPI InetCancel(  
    SOCKET hSocket  
);
```

The **InetCancel** function cancels a blocking operation.

Parameters

hSocket

Handle to the socket.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

When the **InetCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

This function is typically called from within an event handler to signal that the current blocking operation should stop. It may also be used to cancel a blocking operation that is occurring on another thread.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetAbort](#)

InetClientBroadcast Function

```
INT WINAPI InetClientBroadcast(  
    SOCKET hClient,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **InetClientBroadcast** function sends data to all other clients that are connected to the same server.

Parameters

hClient

The socket handle.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server clients.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of clients that the data was sent to. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

The **InetClientBroadcast** function sends the contents of the buffer to all other clients that are connected to the same server as the specified client. This function will block until all clients have been sent a copy of the data. There is no guarantee in which order the clients will receive and process the data that has been broadcast.

This function can only be used with client sessions created as part of the server interface and cannot be used with standard sockets created using the **InetConnect** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetServerBroadcast](#), [InetWrite](#), [InetWriteLine](#)

InetCompareAddress Function

```
BOOL WINAPI InetCompareAddress(  
    LPINTERNET_ADDRESS lpAddress1,  
    LPINTERNET_ADDRESS lpAddress2  
);
```

The **InetCompareAddress** function compares two Internet addresses in a binary format.

Parameters

lpAddress1

A pointer to an INTERNET_ADDRESS structure that contains the first IP address to be compared.

lpAddress2

A pointer to an INTERNET_ADDRESS structure that contains the second IP address to be compared.

Return Value

If the function succeeds and the two addresses are identical, the return value is non-zero. If the function fails or the two addresses are not identical, the return value is zero. If either parameter is NULL, or the address family for the two addresses are not the same, the last error code will be updated. If the addresses are valid and in the same address family, but are not identical, the last error code will be set to NO_ERROR.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetGetHostAddress](#), [InetGetLocalAddress](#), [InetGetPeerAddress](#), [INTERNET_ADDRESS](#)

InetConnect Function

```
SOCKET WINAPI InetConnect(  
    LPCTSTR lpszHostName,  
    UINT nPort,  
    UINT nProtocol,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSECURITYCREDENTIALS lpCredentials  
);
```

The **InetConnect** function is used to establish a connection with a server.

Parameters

lpszHostName

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nPort

The port number the server is listening on. This value must be greater than zero.

nProtocol

The protocol to be used when establishing the connection. This may be one of the following values:

Constant	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This is the default protocol.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. Note that UDP is unreliable in that there is no way for the sender to know that the receiver has actually received the datagram.

nTimeout

The number of seconds to wait for the connection to complete before failing the current operation.

dwOptions

An unsigned integer used to specify one or more socket options. This parameter is constructed by using the bitwise Or operator with any of the following values:

Constant	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_DONTROUTE	This option specifies default routing should not be used. This option should not be specified unless absolutely necessary.
INET_OPTION_KEEPAIVE	This option specifies that packets are to be sent to the remote system when no data is being

	exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_NODELAY	This option disables the Nagle algorithm. By default, small amounts of data written to the socket are buffered, increasing efficiency and reducing network congestion. However, this buffering can negatively impact the responsiveness of certain applications. This option disables this buffering and immediately sends data packets as they are written to the socket.
INET_OPTION_RESERVEDPORT	This option specifies the socket should be bound to an unused port number less than 1024, which is typically reserved for well-known system services. If this option is specified, the process may require administrative privileges.
INET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
INET_OPTION_SECURE	This option specifies that a secure connection should be established with the remote host. The specific version of TLS and other security related options are provided in the <i>lpCredentials</i> parameter. If the <i>lpCredentials</i> parameter is NULL, the connection will default to using TLS 1.2 and the strongest cipher suites available. Older versions of Windows prior to Windows 7 and Windows Server 2008 R2 only support TLS 1.0 and secure connections will automatically downgrade on those platforms.
INET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
INET_OPTION_FREETHREAD	This option specifies the socket returned by this function may be used by any thread, and is not

limited to the thread which created it. The application is responsible for ensuring that access to the socket is synchronized across multiple threads.
--

lpCredentials

A pointer to a [SECURITYCREDENTIALS](#) structure. This parameter is only used if `INET_OPTION_SECURE` is specified for a TCP connection. This parameter may be `NULL`, in which case no client credentials will be provided to the server. If client credentials are required, the fields *dwSize*, *lpszCertStore*, and *lpszCertName* must be defined, while other fields may be left undefined. Set *dwSize* to the size of the **SECURITYCREDENTIALS** structure.

Return Value

If the function succeeds, the return value is a handle to a socket. If the function fails, the return value is `INVALID_SOCKET`. To get extended error information, call **InetGetLastError**.

Remarks

To prevent this function from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **InetConnectEx** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **InetAttachThread** function.

Specifying the `INET_OPTION_FREETHREAD` option enables any thread to call any function using the socket handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the socket is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same socket handle.

It is not recommend that you disable the Nagle algorithm by specifying the `INET_OPTION_NODELAY` flag unless it is absolutely required. Doing so can have a significant, negative impact on the performance of the application and network.

When this function is called with UDP as the specified protocol, it does not actually establish a connection in the same way that a TCP stream connection is created. Instead, it simply establishes a default destination IP address and port that is used with subsequent **InetRead** and **InetWrite** calls.

To enable event notification, use the **InetRegisterEvent** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

InetConnectEx, InetDisableEvents, InetDisconnect, InetInitialize, InetRead, InetRegisterEvent,
InetWrite

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

InetConnectEx Function

```
SOCKET WINAPI InetConnectEx(  
    LPCTSTR lpszHostName,  
    UINT nPort,  
    UINT nProtocol,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPCTSTR lpszLocalAddress,  
    UINT nLocalPort,  
    LPSECURITYCREDENTIALS LpCredentials  
);
```

The **InetConnectEx** function is used to establish a connection with a server.

Parameters

lpszHostName

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nPort

The port number the server is listening on. This value must be greater than zero.

nProtocol

The protocol to be used when establishing the connection. This may be one of the following values:

Constant	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This is the default protocol.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. Note that UDP is unreliable in that there is no way for the sender to know that the receiver has actually received the datagram.

nTimeout

The number of seconds to wait for the connection to complete before failing the operation.

dwOptions

An unsigned integer used to specify one or more socket options. This parameter is constructed by using the bitwise Or operator with any of the following values:

Constant	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.

INET_OPTION_NODELAY	This option disables the Nagle algorithm. By default, small amounts of data written to the socket are buffered, increasing efficiency and reducing network congestion. However, this buffering can negatively impact the responsiveness of certain applications. This option disables this buffering and immediately sends data packets as they are written to the socket.
INET_OPTION_RESERVEDPORT	This option specifies the socket should be bound to an unused port number less than 1024, which is typically reserved for well-known system services. If this option is specified, the process may require administrative privileges.
INET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
INET_OPTION_SECURE	This option specifies that a secure connection should be established with the remote host. The specific version of TLS and other security related options are provided in the <i>lpCredentials</i> parameter. If the <i>lpCredentials</i> parameter is NULL, the connection will default to using TLS 1.2 and the strongest cipher suites available. Older versions of Windows prior to Windows 7 and Windows Server 2008 R2 only support TLS 1.0 and secure connections will automatically downgrade on those platforms.
INET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
INET_OPTION_FREETHREAD	This option specifies the socket returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access

	to the socket is synchronized across multiple threads.
--	--

lpzLocalAddress

A pointer to a string that specifies the local IP address that the socket should be bound to. If this parameter is NULL, then an appropriate address will automatically be used. A specific address should only be used if it is required by the application.

nLocalPort

The local port number that the socket should be bound to. If this parameter is set to zero, then an appropriate port number will automatically be used. A specific port number should only be used if it is required by the application.

lpCredentials

A pointer to a [SECURITYCREDENTIALS](#) structure. This parameter is only used if INET_OPTION_SECURE is specified for a TCP connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields *dwSize*, *lpzCertStore*, and *lpzCertName* must be defined, while other fields may be left undefined. Set *dwSize* to the size of the [SECURITYCREDENTIALS](#) structure.

Return Value

If the function succeeds, the return value is a handle to a socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

To prevent this function from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **InetConnectEx** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **InetAttachThread** function.

Specifying the INET_OPTION_FREETHREAD option enables any thread to call any function using the socket handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the socket is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same socket handle.

It is not recommend that you disable the Nagle algorithm by specifying the INET_OPTION_NODELAY flag unless it is absolutely required. Doing so can have a significant, negative impact on the performance of the application and network.

When this function is called with UDP as the specified protocol, it does not actually establish a connection in the same way that a TCP stream connection is created. Instead, it simply establishes a default destination IP address and port that is used with subsequent **InetRead** and **InetWrite** calls.

To enable event notification, use the **InetRegisterEvent** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cswsock10.h

Import Library: cs wskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetConnect](#), [InetDisableEvents](#), [InetDisconnect](#), [InetInitialize](#), [InetRegisterEvent](#)

InetCreateSecurityCredentials Function

```
BOOL WINAPI InetCreateSecurityCredentials(  
    DWORD dwProtocol,  
    DWORD dwOptions,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName,  
    LPVOID lpvReserved,  
    LPSECURITYCREDENTIALS* lppCredentials  
);
```

The **InetCreateSecurityCredentials** function creates a **SECURITYCREDENTIALS** structure.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is

	supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A pointer to a string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A pointer to a string which specifies the certificate owner's password. A value of NULL specifies

that no password is required. This parameter is only used if a PKCS12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A pointer to a string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

lpvReserved

Pointer reserved for future use. Set it to NULL when using this function.

lppCredentials

Pointer to an [LPSECURITYCREDENTIALS](#) pointer. The memory for the credentials structure will be allocated by this function and must be released by calling the **InetDeleteSecurityCredentials** function when it is no longer needed. The pointer value must be set to NULL before the function is called. It is important to note that this is a pointer to a pointer variable, not a pointer to the SECURITYCREDENTIALS structure itself.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The structure that is created by this function may be used as client credentials when establishing a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can

be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **InetAcceptEx**, **InetAsyncAcceptEx**, **InetAsyncConnectEx** or **InetConnectEx** functions.

Example

```
LPSECURITYCREDENTIALS lpSecCred = NULL;
InetCreateSecurityCredentials(SEcurity_PROTOCOL_DEFAULT,
                             CREDENTIAL_STORE_CURRENT_USER,
                             NULL,
                             NULL,
                             strCertStore,
                             strCertName,
                             NULL,
                             &lpSecCred);

hAcceptSocket = InetAsyncAcceptEx(hListenSocket,
                                  nTimeout,
                                  dwOptions,
                                  lpSecCred,
                                  hEventWnd,
                                  uEventMsg);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetAcceptEx](#), [InetAsyncAcceptEx](#), [InetAsyncConnectEx](#), [InetConnectEx](#),
[InetDeleteSecurityCredentials](#)

InetDeleteSecurityCredentials Function

```
VOID WINAPI InetDeleteSecurityCredentials(  
    LPSECURITYCREDENTIALS* LppCredentials  
);
```

The **InetDeleteSecurityCredentials** function deletes an existing SECURITYCREDENTIALS structure.

Parameters

lppCredentials

Pointer to an LPSECURITYCREDENTIALS pointer. On exit from the function, the pointer value will be NULL.

Return Value

None.

Example

```
if (lpSecCred != NULL)  
    InetDeleteSecurityCredentials(&lpSecCred);
```

Remarks

This function can be used to release the memory allocated to the client or server credentials after a secure connection has been established.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: csoskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetCreateSecurityCredentials](#), [InetUninitialize](#)

InetDetachSocket Function

```
BOOL WINAPI InetDetachSocket(  
    SOCKET hSocket  
    DWORD dwThreadId  
);
```

The **InetDetachSocket** function detaches the specified socket from the current process.

Parameters

hSocket

Handle to the socket.

dwThreadId

The ID of the thread that owns the socket handle. A value of zero specifies that the current thread is the owner.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetDetachSocket** function will release the memory that the library has allocated for the socket without destroying the socket handle. After the function returns, the socket can no longer be used with other functions in the library, however the socket handle remains valid. This is typically used when passing a socket handle between processes, where the parent process detaches the socket prior to creating the child process. The child then calls **InetAttachSocket** to attach the socket handle to its own process.

This function should never be used with a secure socket connection because detaching a secure socket will force the security context for that session to be released. If the socket is attached to another process, it will not have the security context originally created when the connection was established and will be unable to encrypt or decrypt the data stream.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

See Also

[InetAttachThread](#), [InetAttachSocket](#), [InetDisconnect](#)

InetDisableEvents Function

```
INT WINAPI InetDisableEvents(  
    SOCKET hSocket  
);
```

The **InetDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hSocket

The socket handle.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

This function affects both event notification and event callbacks. Any outstanding events in the message queue should be ignored by the application.

This function cannot be used with sockets that are created by the SocketWrench server interface. Those sockets are managed separately in their own thread, and event notifications are handled inside the callback function specified when the server is created using the **InetServerStart** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetEnableEvents](#), [InetFreezeEvents](#), [InetRegisterEvent](#)

InetDisableSecurity Function

```
INT WINAPI InetDisableSecurity(  
    SOCKET hSocket,  
    DWORD dwReserved  
);
```

The **InetDisableSecurity** function disables a secure session with the remote host.

Parameters

hSocket

The socket handle.

dwReserved

Reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetDisableSecurity** function disables a secure session, with subsequent calls to **InetRead** and **InetWrite** sending and receiving unencrypted data. It is important to note that because this function sends a shutdown message to terminate the secure session, this may cause connection to be closed by the remote host.

This function does not close the socket. Use the **InetDisconnect** function to close the socket and release the resources allocated for the current session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: csosk10.lib

See Also

[InetCreateSecurityCredentials](#), [InetDeleteSecurityCredentials](#), [InetEnableSecurity](#)

InetDisableTrace Function

```
BOOL WINAPI InetDisableTrace();
```

The **InetDisableTrace** function disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.
To get extended error information, call **InetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetEnableTrace](#)

InetDisconnect Function

```
INT WINAPI InetDisconnect(  
    SOCKET hSocket  
);
```

The **InetDisconnect** function terminates the connection, closing the socket and releasing the memory allocated for the session.

Parameters

hSocket

The socket handle.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

Once the connection has been terminated, the socket handle is no longer valid and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new connection is established. An application should always consider the socket handle to be opaque and never depend on it being a specific value.

After a socket is closed, it will go into a TIME-WAIT state which prevents an application from using the same source and destination address and port numbers bound to that socket for a brief period of time, typically two to four minutes. This is normal behavior designed to prevent delayed or misrouted packets of data from being read by a subsequent connection. This can have an impact on an application that rapidly connects and disconnects over a short period of time because it can exhaust the pool of ephemeral ports.

If this function is called using a server socket handle returned by the **InetServerStart** function, all active client connections will be disconnected, the listening socket will be closed and the server thread will terminate. If this function is called using a client socket handle that was allocated as part of the SocketWrench server interface, it will terminate the client connection and the thread that manages it.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[InetAsyncConnect](#), [InetAsyncConnectEx](#), [InetConnect](#), [InetConnectEx](#), [InetDisableEvents](#), [InetEnableEvents](#), [InetUninitialize](#)

InetEnableEvents Function

```
INT WINAPI InetEnableEvents(  
    SOCKET hSocket,  
    HWND hEventWnd,  
    UINT nEventMsg  
);
```

The **InetEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **InetRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hSocket

The socket handle.

hEventWnd

Handle to the event notification window. This window receives a user-defined message which specifies the event that has occurred. If this value is NULL, event notification is disabled.

nEventMsg

An unsigned integer which specifies the user-defined message that is sent when an event occurs. This parameter's value must be greater than the value of WM_USER.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the socket handle. One or more of the following event identifiers may be sent:

Constant	Description
INET_EVENT_ACCEPT	The process has received a connection request from a client and should accept the connection using the InetAsyncAccept function. This event is only generated for server applications which have created an asynchronous socket using the InetAsyncListen function.
INET_EVENT_CONNECT	The connection to the remote host has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will also be posted if an error has occurred.
INET_EVENT_DISCONNECT	The remote host has closed the connection. The process should read any remaining data and disconnect.

INET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the process has read at least some of the data from the socket. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_WRITE	The process can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_TIMEOUT	The process has timed-out waiting for a blocking operation to complete. This event is only generated for synchronous sockets.
INET_EVENT_CANCEL	The application has canceled a blocking operation. This event is fired once an operation has been terminated by the InetCancel function, and control has been returned to the calling process.

This function cannot be used with sockets that are created by the SocketWrench server interface. Those sockets are managed separately in their own thread, and event notifications are handled inside the callback function specified when the server is created using the **InetServerStart** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetDisableEvents](#), [InetFreezeEvents](#), [InetRegisterEvent](#)

InetEnableSecurity Function

```
INT WINAPI InetEnableSecurity(  
    SOCKET hSocket,  
    DWORD dwOptions,  
    LPSECURITYCREDENTIALS lpCredentials  
);
```

The **InetEnableSecurity** function enables a secure session with the remote host.

Parameters

hSocket

The socket handle.

dwOptions

An unsigned integer value that specifies additional security options. It may have a value of zero or one of the following options:

Constant	Description
INET_SECURE_CLIENT	The certificate specified by the <i>lpCredentials</i> parameter will be used as a client certificate, and the application will begin to negotiate the secure session as a client by initiating the handshake with the server. The certificate that is used must be a valid client certificate with a private key associated with it. If the <i>lpCredentials</i> parameter is NULL, then a secure client session will be initiated without a client certificate.
INET_SECURE_SERVER	The certificate specified by the <i>lpCredentials</i> parameter will be used as a server certificate, and the application will wait for the remote host to initiate the handshake that establishes the parameters of the secure session. The certificate that is used must be a valid server certificate and have a private key associated with it. The <i>lpCredentials</i> parameter cannot be NULL if this option is specified.

lpCredentials

Pointer to a SECURITYCREDENTIALS structure. This parameter may be NULL, in which case no client credentials will be provided. If client credentials are required, the fields *dwSize*, *lpzCertStore*, and *lpzCertName* must be defined, while other fields may be left undefined. Set *dwSize* to the size of the SECURITYCREDENTIALS structure.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetEnableSecurity** function enables a secure communications session with the remote host, automatically negotiating the encryption algorithm and validating the certificate. This function is useful if the application needs to establish a standard connection to the remote host and then negotiate a secure connection at a later point. If the function succeeds, all subsequent calls to **InetRead** and **InetWrite** to receive and send data will be encrypted.

If the *dwOptions* parameter has a value of zero and the socket was created using **InetConnect** or related functions to establish a client connection, then **InetEnableSecurity** will initiate the handshake with the remote host to establish a secure session. If the **InetAccept** or related functions were used to accept a connection from a client, then the function will block and wait for the client to initiate the handshake.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

See Also

[InetCreateSecurityCredentials](#), [InetDeleteSecurityCredentials](#), [InetDisableSecurity](#)

InetEnableTrace Function

```
BOOL WINAPI InetEnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **InetEnableTrace** function enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

Name of the trace log file. If this parameter is NULL or empty, the file CSTRACE.LOG is used. The directory for CSTRACE.LOG is given by the TEMP environment variable, if it is defined; otherwise, the directory given by the TMP environment variable is used, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Constant	Description
0	TRACE_DEFAULT	All function calls are written to the trace file. This is the default value.
1	TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
2	TRACE_WARNING	Only those function calls which fail, or return values which indicate a warning, are recorded in the trace file.
4	TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call [InetGetLastError](#).

Remarks

When trace logging is enabled, the logfile is opened, appended to and closed for each socket function call. Using the same logfile name, you can do the same in your application to add additional information to the logfile if needed. This can provide an application-level context for the entries made by the library. Make sure that the logfile is closed after the data has been written.

The TRACE_HEXDUMP option can produce very large logfiles, since all data that is being sent and received by the application is logged. To reduce the size of the file, you can enable and disable logging around limited sections of code that you wish to analyze.

All of the SocketTools networking components that use the Windows Sockets API support logging. If you are using multiple components, you only need to enable tracing once in your application or once per thread in a multithreaded application.

To redistribute an application that includes logging functionality, the **cstrcv10.dll** library must be included as part of the installation package. This library provides the trace logging features, and if

it is not available the **InetEnableTrace** function will fail. Note that this is a standard Windows DLL and does not need to be registered, it only needs to be redistributed with your application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetDisableTrace](#)

InetEnumHostAliases Function

```
INT WINAPI InetEnumHostAliases(  
    LPCTSTR LpszHostName,  
    LPTSTR LpszBuffer,  
    INT cbMaxBuffer  
);
```

The **InetEnumHostAliases** function returns a byte array that contains all the aliases for a specified host.

Parameters

lpszHostName

Host name

lpszBuffer

Buffer where aliases are to be written. The first name written to the buffer is the primary host name. Each name or alias is terminated by a null, followed by the next alias, if there is one.

cbMaxBuffer

Maximum number of bytes to be written to *lpszBuffer*.

Return Value

If the function succeeds, the return value is the number of bytes written to *lpszBuffer*. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

InetEnumNetworkAddresses Function

```
INT WINAPI InetEnumNetworkAddresses(  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddressList,  
    INT nMaxAddresses  
);
```

The **InetEnumNetworkAddresses** function returns the list of network addresses that are configured for the local host.

Parameters

nAddressFamily

An integer which identifies the type of IP address that should be returned by this function. It may be one of the following values:

Constant	Description
INET_ADDRESS_ANY	Return both IPv4 or IPv6 addresses assigned to the local host, depending on how the system is configured and which network interfaces are enabled. This option is only recommended for applications that require support for IPv6 connections.
INET_ADDRESS_IPV4	Return only the IPv4 addresses assigned to the local host. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Return only the IPv6 addresses assigned to the local host. All bytes in the <i>ipNumber</i> array are significant. This option is only recommended for those applications that require support for IPv6 connections.

lpAddressList

A pointer to an array of [INTERNET_ADDRESS](#) structures that will contain the IP address of each local network interface. This parameter may be NULL, in which case the method will only return the number of available addresses.

nMaxAddresses

Maximum number of addresses to be returned. If the *lpAddressList* parameter is NULL, this value must be zero.

Return Value

If the function succeeds, the return value is the number of network addresses that are configured for the local host. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

If the *nAddressFamily* parameter is specified as INET_ADDRESS_ANY, the application must be prepared to accept IPv6 addresses returned by this function. On Windows Vista and later versions of the operating system, IPv6 support is enabled and the local network adapter will have IPv6 addresses assigned to them by default. For legacy applications that only recognize IPv4 addresses,

the *nAddressFamily* parameter should always be specified as INET_ADDRESS_IPV4 to ensure that only IPv4 addresses are returned.

This function will ignore addresses that are bound to a disabled interface, as well as those addresses bound to a virtual loopback interface. For example, although the loopback address 127.0.0.1 is a valid network address, it will not be included in list of addresses returned by this function.

The first IPv4 or IPv6 address returned by this function is typically the address assigned to the primary network adapter on the local system. However, your application should not depend on addresses being returned in any particular order. If the system has dial-up networking or virtualization software installed, this function may also include the IP addresses assigned to any virtualized network adapters installed by that software.

Example

```
INTERNET_ADDRESS *lpAddressList = NULL;
INT nAddressCount = InetEnumNetworkAddresses(INET_ADDRESS_IPV4, NULL, 0);

if (nAddressCount > 0)
{
    // Allocate memory for the array of IP addresses
    lpAddressList = (INTERNET_ADDRESS *)LocalAlloc(LPTR, nAddressCount *
sizeof(INTERNET_ADDRESS));

    if (lpAddressList == NULL)
    {
        // Virtual memory exhausted
        return;
    }

    // Populate the array with the addresses
    nAddressCount = InetEnumNetworkAddresses(INET_ADDRESS_IPV4, lpAddressList,
nAddressCount);
}

_tprintf(_T("There are %d local network addresses assigned\n"), nAddressCount);

// Display each IP address assigned to the local system
for (INT nIndex = 0; nIndex < nAddressCount; nIndex++)
{
    TCHAR szValue[64];

    // Convert the IP address to a printable string
    InetFormatAddress(lpAddressList + nIndex, szValue, 64);
    _tprintf(_T("%d: %s\n"), nIndex, szValue);
}

// Free the memory allocated for the IP address list
if (lpAddressList != NULL)
    LocalFree((HLOCAL)lpAddressList);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetGetAdapterAddress](#), [InetGetHostAddress](#), [InetGetLocalAddress](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

InetEnumServerClients Function

```
INT WINAPI InetEnumServerClients(  
    SOCKET hServer,  
    SOCKET * lpClients,  
    INT nMaxClients  
);
```

The **InetEnumServerClients** function returns a list of active client sessions established with the specified server.

Parameters

hServer

Handle to the server socket.

lpClients

Pointer to an array of socket handles which identifies all client connections. If this parameter is NULL, then the function will return the number of active client connections established with the server.

nMaxClients

Maximum number of client socket handles to be returned. If the *lpClients* parameter is NULL, this parameter should be specified with a value of zero.

Return Value

If the function succeeds, the return value is the number of active client connections to the server. A return value of zero indicates that there are no active client sessions. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

If the *nMaxClients* parameter is less than the number of active client connections, the function will fail. To dynamically determine the number of active connections, call the function with the *lpClients* parameter with a value of NULL, and the *nMaxClients* parameter with a value of zero. To enumerate the active clients that match a specific IP address, use the **InetEnumServerClientsByAddress** function.

This function will not enumerate clients that have disconnected from the server, even if the session thread is still active. If the server is in the process of shutting down, this function will return zero, indicating no active client sessions, even though there may be clients that are still in the process of disconnecting from the server. To determine the actual number of client sessions regardless of their status, use the **InetGetClientThreads** function.

The socket handle for the server must be one that was created using the **InetServerStart** function, and cannot be a socket that was created using **InetListen** or **InetListenEx**.

Example

```
INT nMaxClients = InetEnumServerClients(hServer, NULL, 0);  
  
if (nMaxClients > 0)  
{  
    SOCKET *lpClients = NULL;  
  
    // Allocate memory for client sockets  
    lpClients = (SOCKET *)LocalAlloc(LPTR, nMaxClients * sizeof(SOCKET));
```

```
if (lpClients == NULL)
{
    // Virtual memory has been exhausted
    return;
}

nMaxClients = InetEnumServerClients(hServer, lpClients, nMaxClients);
if (nMaxClients == INET_ERROR)
{
    // Unable to obtain list of connected clients
    return;
}

for (INT nClient = 0; nClient < nMaxClients; nClient++)
{
    // Perform some action with each client socket
    SOCKET hClient = lpClients[nClient];
}

// Free memory allocated for client sockets
LocalFree((HLOCAL)lpClients);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetEnumServerClientsByAddress](#), [InetGetClientThreads](#), [InetServerBroadcast](#), [InetServerStart](#)

InetEnumServerClientsByAddress Function

```
INT WINAPI InetEnumServerClientsByAddress(  
    SOCKET hServer,  
    LPINTERNET_ADDRESS LpAddress,  
    SOCKET * LpClients,  
    INT nMaxClients  
);
```

The **InetEnumServerClientsByAddress** function returns a list of active client sessions established with the specified server that match the specified IP address.

Parameters

hServer

Handle to the server socket.

lpAddress

Pointer to an INTERNET_ADDRESS structure that specifies the client IP address. If this parameter is NULL, then all active client sessions will be enumerated.

lpClients

Pointer to an array of socket handles which identifies all client connections that match the specified IP address. If this parameter is NULL, then the function will return the number of active client connections established with the server.

nMaxClients

Maximum number of client socket handles to be returned. If the *lpClients* parameter is NULL, this parameter should be specified with a value of zero.

Return Value

If the function succeeds, the return value is the number of active client connections to the server that match the specified IP address. A return value of zero indicates that no clients have connected from the specified IP address. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

If the *nMaxClients* parameter is less than the number of active client connections that match the specified address, the function will fail. To dynamically determine the number of active connections, call the function with the *lpClients* parameter with a value of NULL, and the *nMaxClients* parameter with a value of zero. If the *lpAddress* parameter is NULL, this function will return a list of all active clients connected to the server and is effectively the same as calling the **InetEnumServerClients** function. If an address is provided, it must be a valid IP address in a supported address family.

This function will not enumerate clients that have disconnected from the server, even if the session thread is still active. If the server is in the process of shutting down, this function will return zero, indicating no active client sessions, even though there may be clients that are still in the process of disconnecting from the server.

The socket handle for the server must be one that was created using the **InetServerStart** function, and cannot be a socket that was created using **InetListen** or **InetListenEx**.

Example

```
INTERNET_ADDRESS ipAddress;
```

```

if (InetGetPeerAddress(hSocket, &ipAddress, NULL) == INET_ERROR)
{
    // Unable to get the peer IP address for the socket
    return;
}

// Determine the number of clients that have the same IP address
INT nMaxClients = InetEnumServerClientsByAddress(hServer, &ipAddress, NULL, 0);

if (nMaxClients > 0)
{
    SOCKET *lpClients = NULL;

    // Allocate memory for client sockets
    lpClients = (SOCKET *)LocalAlloc(LPTR, nMaxClients * sizeof(SOCKET));
    if (lpClients == NULL)
    {
        // Virtual memory has been exhausted
        return;
    }

    nMaxClients = InetEnumServerClientsByAddress(hServer, &ipAddress, lpClients,
nMaxClients);
    if (nMaxClients == INET_ERROR)
    {
        // Unable to obtain list of connected clients
        return;
    }

    for (INT nClient = 0; nClient < nMaxClients; nClient++)
    {
        // Perform some action with each client socket
        SOCKET hClient = lpClients[nClient];
    }

    // Free memory allocated for client sockets
    LocalFree((HLOCAL)lpClients);
}

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetEnumServerClients](#), [InetGetClientThreads](#), [InetServerBroadcast](#), [InetServerStart](#)

InetEventProc Function

```
VOID CALLBACK InetEventProc(  
    SOCKET hSocket,  
    UINT nEvent,  
    DWORD dwError,  
    DWORD_PTR dwParam  
);
```

The **InetEventProc** function is an application-defined callback function that processes events generated by the calling process.

Parameters

hSocket

The socket handle.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the [InetRegisterEvent](#) function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **InetRegisterEvent** function. The **InetEventProc** function is a placeholder for the application-defined function name.

If the callback function is being used with the **InetServerStart** function, the function will be called in the context of the thread that is currently managing the server or client session. You must ensure that any access to global or static variables is synchronized, otherwise the results may be unpredictable. It is recommended that you do not declare any static variables within the callback function itself. If you need to manage state information for a specific client, then use the **InetGetClientData** and **InetSetClientData** functions which will allow you to access application defined data for that client session in a thread-safe manner.

When this callback function is used for event notifications from the server interface, the the *hSocket* parameter specifies the client socket handle, except for the INET_EVENT_ACCEPT event, in which case the handle references the server socket handle. To obtain the handle of the client connection that was just accepted, use the **InetGetServerClient** function. To obtain the handle to the server using the client socket handle, use the **InetGetClientServer** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetDisableEvents](#), [InetEnableEvents](#), [InetFreezeEvents](#), [InetGetClientServer](#), [InetGetServerClient](#)

InetFindClientMoniker Function

```
SOCKET WINAPI InetFindClientMoniker(  
    SOCKET hServer,  
    LPCTSTR LpszMoniker  
);
```

The **InetFindClientMoniker** function returns a handle to the client socket which matches the specified moniker.

Parameters

hServer

A handle to the server.

lpszMoniker

A pointer to a string which specifies the client moniker to search for. This parameter cannot be NULL and cannot specify an empty string.

Return Value

If the function succeeds, the return value is the handle to the client socket for the session that matches the specified moniker. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

A client moniker is a string which can be used to uniquely identify a specific client session aside from its socket handle. A moniker can be assigned to the client session using the **InetSetClientMoniker** function. This function will search all active client sessions for the server, and returns the socket handle to the client that matches the specified moniker. If there is no match, an error will be returned.

The moniker can be any string value, however monikers are not case sensitive and may not contain embedded null characters. The maximum length of a moniker is 127 characters.

The socket handle for the server must be one that was created using the **InetServerStart** function, and cannot be a socket that was created using **InetListen** or **InetListenEx**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetGetClientMoniker](#), [InetGetClientServer](#), [InetGetClientThreadId](#), [InetSetClientMoniker](#)

InetFlush Function

```
INT WINAPI InetFlush(  
    SOCKET hSocket  
);
```

The **InetFlush** function flushes the internal send and receive buffers used by the socket.

Parameters

hSocket

The socket handle.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetFlush** function will flush any data waiting to be read or written to the remote host . It is important to note that this function is not similar to flushing data to a disk file; it does not ensure that a specific block of data has been written to the socket. For example, you should never call this function immediately after calling **InetWrite** or prior to calling **InetDisconnect**.

An application never needs to use **InetFlush** under normal circumstances. This function is only to be used when the application needs to immediately return the socket to an inactive state with no pending data to be read or written. Calling this function may result in data loss and should only be used if you understand the implications of discarding any data which has been sent by the remote host.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetIsReadable](#), [InetIsWritable](#), [InetPeek](#), [InetRead](#), [InetWrite](#)

InetFormatAddress Function

```
INT WINAPI InetFormatAddress(  
    LPINTERNET_ADDRESS LpAddress,  
    LPTSTR LpszAddress,  
    INT cchAddress  
);
```

The **InetFormatAddress** function converts a numeric IP address to a printable string. The format of the string depends on whether an IPv4 or IPv6 address is specified.

Parameters

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure which specifies the numeric IP address that should be converted to a string.

lpszAddress

A pointer to a string buffer that will contain the formatted IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length.

cchAddress

The maximum number of characters that can be copied into the address buffer.

Return Value

If the function succeeds, the return value is the length of the IP address string. If the function fails, the return value is `INET_ERROR`, meaning that the IP address could not be converted into a string. Typically this indicates that the pointer to the `INTERNET_ADDRESS` structure is invalid, or the data does not specify a valid IP address family.

Remarks

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetGetExternalAddress](#), [InetGetHostAddress](#), [InetGetLocalAddress](#), [InetGetPeerAddress](#), [INTERNET_ADDRESS](#)

InetFreezeEvents Function

```
INT WINAPI InetFreezeEvents(  
    SOCKET hSocket,  
    BOOL bFreeze  
);
```

The **InetFreezeEvents** function is used to suspend and resume event handling.

Parameters

hSocket

Socket handle.

bFreeze

A non-zero value specifies that event handling should be suspended. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if event handling has been suspended, and four read events occur, only one of those read events will be posted to the application when even handling is resumed. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

See Also

[InetDisableEvents](#), [InetEnableEvents](#), [InetRegisterEvent](#)

InetGetAdapterAddress Function

```
INT WINAPI InetGetAdapterAddress(  
    INT nAdapterIndex,  
    INT nAddressType,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

Return the IP or MAC assigned to the specified network adapter.

Parameters

nAdapterIndex

An integer value that identifies the network adapter.

nAddressType

An integer value which specifies the type of address that should be returned:

Constant	Description
INET_ADAPTER_IPV4	The address string will contain the primary IPv4 unicast address assigned to the network adapter.
INET_ADAPTER_IPV6	The address string will contain the primary IPv6 unicast address assigned to the network adapter.
INET_ADAPTER_MAC	The address string will contain the media access control (MAC) address assigned to the network adapter.

lpszAddress

A string buffer that will contain the IP or MAC address assigned to the adapter. This parameter cannot be NULL and it is recommended that it be at least 64 characters in length to provide enough space for any address type.

nMaxLength

The maximum number of characters that can be copied into the string buffer, including the terminating null character. If the buffer is too small to store the complete address, this function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied to the string buffer, not including the terminating null character. A return value of zero indicates that the requested address type has not been assigned to the adapter. If the function fails, the return value is `INET_ERROR` and this typically indicates that either the adapter index is invalid or the string buffer is not large enough to store the complete address. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetAdapterAddress** function will return the IPv4, IPv6 or MAC address assigned to a specific network adapter. The primary network adapter has an index value of zero, and it increments for each adapter that is configured on the local system.

The media access control (MAC) address is a 48 bit or 64 bit value that is assigned to each network interface and is used for identification and access control. All network devices on the

same subnet must be assigned their own unique MAC address. Unlike IP addresses which may be assigned dynamically and can be frequently changed, MAC addresses are considered to be more permanent because they are usually assigned by the device manufacturer and stored in firmware. Note that in some cases it is possible to change the address assigned to a device, and virtual network interfaces may have configurable MAC addresses.

This function returns the MAC address string as sequence of hexadecimal values separated by a colon. An example of a 48 bit MAC address would be "01:23:45:67:89:AB". Note that some virtual network adapters may not have a MAC address assigned to them, in which case this function would return zero.

This function will ignore network adapters that have been disabled, as well as those that are bound to a virtual loopback interface. If the system has dial-up networking or virtualization software installed, this function may also return IP addresses assigned to a virtualized network adapters installed by that software.

Example

```
// Display the IPv4 address assigned to each network adapter
for (INT nIndex = 0;; nIndex++)
{
    TCHAR szAddress[64];
    INT cchAddress;

    cchAddress = InetGetAdapterAddress(nIndex, INET_ADAPTER_IPV4, szAddress,
64);

    if (cchAddress == INET_ERROR)
        break;

    _tprintf(_T("Adapter %d: %s\n"), nIndex, szAddress);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetEnumNetworkAddresses](#), [InetGetLocalAddress](#), [InetGetLocalName](#)

InetGetAddress Function

```
INT WINAPI InetGetAddress(  
    LPCTSTR LpszAddress,  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS LpAddress  
);
```

The **InetGetAddress** function converts an IP address string to binary format.

Parameters

LpszAddress

A pointer to a null terminated string which specifies an IP address. This function recognizes the format for both IPv4 and IPv6 format addresses.

nAddressFamily

An integer which identifies the type of IP address specified by the *LpszAddress* parameter. It may be one of the following values:

Constant	Description
INET_ADDRESS_UNKNOWN	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on the value of the <i>LpszAddress</i> parameter.
INET_ADDRESS_IPV4	Specifies that the address should be in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero. If the <i>LpszAddress</i> parameter does not specify a valid IPv4 address string, this function will fail.
INET_ADDRESS_IPV6	Specifies that the address should be in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. If the <i>LpszAddress</i> parameter does not specify a valid IPv6 address string, this function will fail.

LpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the IP address.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

If the *nAddressFamily* parameter is specified as INET_ADDRESS_UNKNOWN, the application must be prepared to handle IPv6 addresses because it is possible that an IPv6 address string has been specified. For legacy applications that only recognize IPv4 addresses, the *nAddressFamily* member should always be specified as INET_ADDRESS_IPV4 to ensure that only IPv4 addresses are returned and any attempt to specify an IPv6 address string would result in an error.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **InetIsProtocolAvailable** function. If an IPv6 stack is not installed, this function will

fail if the *lpszAddress* parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetFormatAddress](#), [InetIsAddressNull](#), [InetIsAddressRoutable](#), [InetIsProtocolAvailable](#),
[INTERNET_ADDRESS](#)

InetGetAddressFamily Function

```
INT WINAPI InetGetAddressFamily(  
    LPCTSTR lpszAddress,  
);
```

Return the address family for the specified IP address.

Parameters

lpszAddress

A pointer to a string which specifies an IP address. This function recognizes the format for both IPv4 and IPv6 format addresses.

Return Value

If the function succeeds, the return value is the address family for the specified IP address and may be one of the values listed below. If the function fails, the return value is INET_ADDRESS_UNKNOWN. To get extended error information, call **InetGetLastError**.

Constant	Description
INET_ADDRESS_IPV4	The address passed to the function is a valid IPv4 address.
INET_ADDRESS_IPV6	The address passed to the function is a valid IPv6 address.

Remarks

The **InetGetAddressFamily** function returns the address family associated with the specified IP address string. This can be used to determine if a string specifies a valid IPv4 or IPv6 address that can be passed to other functions such as **InetConnect**. Note that this function will not attempt to resolve hostnames, it will only accept IP addresses.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **InetIsProtocolAvailable** function. If an IPv6 stack is not installed, this function will fail if the *lpszAddress* parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetIsAddressNull](#), [InetIsAddressRoutable](#), [InetIsProtocolAvailable](#), [INTERNET_ADDRESS](#)

InetGetBlockingSocket Function

```
SOCKET InetGetBlockingSocket();
```

The **InetGetBlockingSocket** function returns the handle for the socket in the current thread which is currently blocking, if there is one.

Parameters

None.

Return Value

If the function succeeds, the return value is the handle for the socket in the current thread which is currently blocking. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

See Also

[InetAbort](#), [InetCancel](#)

InetGetClientData Function

```
BOOL WINAPI InetGetClientData(  
    SOCKET hClient,  
    LPVOID * lppvData  
);
```

The **InetGetClientData** function returns the application defined data associated with the specified client session.

Parameters

hSocket

The socket handle.

lppvData

Pointer to a void pointer which will contain an application defined value associated with the client session.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates that application defined data for the client session could not be retrieved. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetClientData** function is used to retrieve the application defined data that was previously associated with a client session using the **InetSetClientData** function. This is typically used to associate a pointer to a data structure or a class instance with a specific client handle.

This function can only be used with client socket handles created using the server interface. It cannot be used with socket handles created using the **InetConnect** or **InetAccept** functions. If the socket handle is invalid, or does not reference a client socket handle created by the server, the *lppvData* pointer passed to this function will be initialized to a value of NULL and the function will return a value of zero.

If this function is called with a valid socket handle and there is no data associated with the socket, the function will return a non-zero value and the *lppvData* pointer will be returned with a NULL value. Before dereferencing the pointer returned by this function, the application should always check the return value to ensure the function succeeded and make sure that the pointer is not NULL.

Example

```
UINT *pnValue1 = (UINT *)LocalAlloc(LPTR, sizeof(UINT));  
UINT *pnValue2 = NULL;  
  
*pnValue1 = 1234;  
  
if (InetSetClientData(hSocket, pnValue1) == FALSE)  
{  
    // Unable to associate the data with this session  
    return;  
}  
  
if (InetGetClientData(hSocket, &pnValue2) == FALSE)  
{
```

```
    // Unable to retrieve the data associated with this session
    return;
}

// *pnValue2 == 1234
printf("The value of user defined data is %u\n", *pnValue2);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetGetServerData](#), [InetSetClientData](#), [InetSetServerData](#)

InetGetClientHandle Function

```
SOCKET WINAPI InetGetClientHandle(  
    SOCKET hServer,  
    UINT nClientId  
);
```

The **InetGetClientHandle** function returns the handle for a specific client session based on its ID number.

Parameters

hServer

Handle to the server socket.

nClientId

An unsigned integer value which uniquely identifies the client session.

Return Value

If the function succeeds, the return value is the socket handle for the specified client session. If the function fails, the return value is `INVALID_SOCKET`. To get extended error information, call **InetGetLastError**.

Remarks

Each client connection that is accepted by the server is assigned a unique numeric value. This value can be obtained by calling the **InetGetClientId** function and used by the application to identify that client session. The **InetGetClientHandle** function can then be used to obtain the client socket handle for the session, based on that client ID.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetGetClientId](#), [InetGetClientMoniker](#), [InetSetClientMoniker](#), [InetGetServerClient](#), [InetGetThreadClient](#)

InetGetClientId Function

```
UINT WINAPI InetGetClientId(  
    SOCKET hClient  
);
```

The **InetGetClientId** function returns the unique ID number assigned to the specified client session.

Parameters

hClient

Handle to the client socket.

Return Value

If the function succeeds, the return value is an unsigned integer value which uniquely identifies the client session. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

Each client connection that is accepted by the server is assigned a unique numeric value. This value can be obtained by calling the **InetGetClientId** function and used by the application to identify that client session. The **InetGetClientHandle** function can then be used to obtain the client socket handle for the session, based on that client ID. It is important to note that the actual value of the client ID should be considered opaque. It is only guaranteed that the value will be greater than zero, and that it will be unique to the client session.

While it is possible for a client socket handle to be reused by the operating system, client IDs are unique throughout the life of the server session and are never duplicated.

The socket handle for the client must be one that was created as part of the SocketWrench server interface, and cannot be a socket that was created using the **InetConnect** or **InetAccept** functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

See Also

[InetGetClientHandle](#), [InetGetClientMoniker](#), [InetSetClientMoniker](#), [InetGetServerClient](#)

InetGetClientIdleTime Function

```
DWORD WINAPI InetGetClientIdleTime(  
    SOCKET hClient  
);
```

Returns the number of milliseconds that the specified client session has been idle.

Parameters

hClient

Handle to the client socket.

Return Value

If the function succeeds, the return value is an unsigned integer value which specifies the number of milliseconds the client session has been idle. If the function fails, the return value is INFINITE. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetClientIdleTime** function will return the number of milliseconds that have elapsed since data was exchanged with the client. The elapsed time is limited to the resolution of the system timer, which is typically in the range of 10 milliseconds to 16 milliseconds.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetGetClientHandle](#), [InetGetClientMoniker](#), [InetGetServerClient](#)

InetGetClientMoniker Function

```
INT WINAPI InetGetClientMoniker(  
    SOCKET hSocket,  
    LPTSTR LpszMoniker,  
    INT nMaxLength  
);
```

The **InetGetClientMoniker** function returns the moniker associated with the specified client session.

Parameters

hSocket

Handle to the client socket.

lpszMoniker

Pointer to a string buffer that will contain the moniker for the specified client session when the function returns.

nMaxLength

The maximum number of characters that may be copied into the string buffer. The buffer must be large enough to store the moniker and a terminating null character. The maximum length of a moniker is 127 characters.

Return Value

If the function succeeds, the return value is the number of characters in the moniker string. A return value of zero specifies that no moniker was assigned to the socket. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

A client moniker is a string which can be used to uniquely identify a specific client session aside from its socket handle. A moniker can be assigned to the client session using the **InetSetClientMoniker** function. This function will return the moniker that was previously assigned to the client, if any. To obtain the socket handle associated with a given moniker, use the **InetFindClientMoniker** function.

The socket handle for the client must be one that was created as part of the SocketWrench server interface, and cannot be a socket that was created using the **InetConnect** or **InetAccept** functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetFindClientMoniker](#), [InetGetClientHandle](#), [InetGetClientId](#), [InetSetClientMoniker](#)

InetGetClientPriority Function

```
INT WINAPI InetGetClientPriority(  
    SOCKET hClient  
);
```

The **InetGetClientPriority** function returns the current priority for the specified client session.

Parameters

hClient

Handle to the client socket.

Return Value

If the function succeeds, the return value is the priority for the specified client session. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetClientPriority** function can be used to determine the current priority assigned to the specified client session. The client priority is inherited from the priority specified when the server is started using the **InetServerStart** function. It may be one of the following values:

Constant	Description
INET_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. It is recommended that most applications use this priority.
INET_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the client session. It is typically used with lightweight services running in the background that are designed for few client connections. The client thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
INET_PRIORITY_LOW	This priority lowers the overall resource utilization for the client session and meters the processor utilization for the client session. The client thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
INET_PRIORITY_HIGH	This priority increases the overall resource utilization for the client session and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
INET_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The client thread will be given higher scheduling priority and will be more responsive to network events. It is not recommended that this priority be used on a system with a single processor.

The socket handle for the client must be one that was created as part of the SocketWrench server interface, and cannot be a socket that was created using the **InetConnect** or **InetAccept**

functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: csoskv10.lib

See Also

[InetGetServerPriority](#), [InetServerStart](#), [InetSetClientPriority](#), [InetSetServerPriority](#)

InetGetClientServer Function

```
SOCKET WINAPI InetGetClientServer(  
    SOCKET hClient  
);
```

The **InetGetClientServer** function returns a socket handle to the server for the specified client socket.

Parameters

hClient

Handle to the client socket.

Return Value

If the function succeeds, the return value is the handle to the server that created the client session. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetClientServer** function returns the handle to the server that created the client session. The **InetGetClientServerById** function can be used to obtain the server handle using the client session ID rather than the client socket handle.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[InetGetClientHandle](#), [InetGetClientId](#), [InetGetClientServerById](#)

InetGetClientServerById Function

```
SOCKET WINAPI InetGetClientServerById(  
    UINT nClientId  
);
```

The **InetGetClientServerById** function returns a socket handle to the server for the specified client session identifier.

Parameters

nClientId

Client session identifier.

Return Value

If the function succeeds, the return value is the handle to the server that created the client session. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetClientServerById** function returns the handle to the server that created the client session using the client's unique identifier. The **InetGetClientServer** function can be used to obtain the server handle using the client socket handle rather than the client session ID. This function is typically used in conjunction with the INET_NOTIFY_CONNECT notification message to obtain the handle to the server that generated the event using the client ID passed in the *wParam* message parameter.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[InetGetClientHandle](#), [InetGetClientId](#), [InetGetClientServer](#), [InetServerAsyncNotify](#)

InetGetClientThreadId Function

```
DWORD WINAPI InetGetClientThreadId(  
    SOCKET hClient  
);
```

The **InetGetClientThreadId** function returns the thread ID for the specified client session.

Parameters

hClient

Handle to the client socket.

Return Value

If the function succeeds, the return value is an unsigned integer value which identifies the thread that was created to manage the client session. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The thread ID returned by this function can be used with the **OpenThread** function to obtain a handle to the thread. Until the thread terminates, the thread identifier uniquely identifies the thread throughout the system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetGetClientPriority](#), [InetGetServerThreadId](#), [InetGetThreadClient](#), [InetSetClientPriority](#)

InetGetClientThreads Function

```
INT WINAPI InetGetClientThreads(  
    SOCKET hServer  
);
```

Returns the number of client session threads created by the server.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is the number of client session threads that have been created by the server. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetClientThreads** function returns the number of threads that are managing client sessions for the specified server. If there are no clients connected to the server, this function will return a value of zero. Because this function returns the number of session threads, the value returned will include those clients that are in the process of disconnecting from the server but their session thread has not yet terminated. This differs from the **InetEnumServerClients** function which will only enumerate active clients.

If you wish to determine when the last client has disconnected from the server, call this function within an event handler for the `INET_EVENT_DISCONNECT` event. If the function returns a value greater than one, then there are other client sessions that are either connected or in the process of terminating.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

See Also

[InetEnumServerClients](#), [InetEnumServerClientsByAddress](#)

InetGetDefaultHostFile Function

```
INT WINAPI InetGetDefaultHostFile(  
    LPTSTR lpszFileName,  
    INT nMaxLength  
);
```

The **InetGetDefaultHostFile** function returns the fully qualified path name of the host file on the local system. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **InetGetHostAddress** and **InetGetHostNames** function. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

```
ipaddress hostname [hostalias ...]
```

For example, one typical entry maps the name "localhost" to the local loopback IP address. This would be entered as:

```
127.0.0.1 localhost
```

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

The location of the default host file depends on the operating system. For Windows 95/98 and Windows Me the file is stored in C:\Windows\hosts and for Windows NT and later versions the file is stored in C:\Windows\system32\drivers\etc\hosts. Regardless of platform, there is no filename extension and this file may or may not exist on a given system.

Parameters

lpszFileName

Pointer to a string buffer that will contain the fully qualified file name to the default host file. It is recommended that this buffer be at least MAX_PATH characters in size. This parameter may be NULL, in which case the function will return the length of the string, not including the terminating null byte.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the function succeeds, the return value is length of the string. A return value of zero indicates that the default host file could not be determined for the current platform. To get extended error information, call **InetGetLastError**.

Remarks

This function only returns the default location of the host file and does not determine if the file actually exists. It is not required that a host file be present on the system.

The default host file is processed before performing a nameserver lookup when resolving a hostname into an IP address, or an IP address into a hostname.

To specify an alternate local host file, use the **InetSetHostFile** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cswsock10.h

Import Library: cswskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetGetHostAddress](#), [InetGetHostFile](#), [InetGetHostName](#), [InetSetHostFile](#)

InetGetErrorString Function

```
INT WINAPI InetGetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);
```

The **InetGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the [InetGetLastError](#) function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The last-error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is 0, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetGetLastError](#), [InetSetLastError](#)

InetGetExternalAddress Function

```
INT WINAPI InetGetExternalAddress(  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddress  
);
```

The **InetGetExternalAddress** function returns the external IP address for the local system.

Parameters

nAddressFamily

An integer which identifies the type of IP address that should be returned by this function. It may be one of the following values:

Constant	Description
INET_ADDRESS_IPV4	Specifies that the address should be in IPv4 format. The method will attempt to determine the external IP address using an IPv4 network connection.
INET_ADDRESS_IPV6	Specifies that the address should be in IPv6 format. The method will attempt to determine the external IP address using an IPv6 network connection and requires that the local host have an IPv6 network interface installed and enabled.

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the external IP address of the local host.

Return Value

If the function succeeds, the return value is zero and the [INTERNET_ADDRESS](#) structure contains the external IP address for the local host in binary form. If the function fails, the return value is [INET_ERROR](#). To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetExternalAddress** function returns the IP address assigned to the router that connects the local host to the Internet. This is typically used by an application executing on a system in a local network that uses a router which performs Network Address Translation (NAT). In that network configuration, the **InetGetLocalAddress** function will only return the IP address for the local system on the LAN side of the network unless a connection has already been established to a remote host. The **InetGetExternalAddress** function can be used to determine the IP address assigned to the router on the Internet side of the connection and can be particularly useful for servers running on a system behind a NAT router.

This function requires that you have an active connection to the Internet and calling this function on a system that uses dial-up networking may cause the operating system to automatically connect to the Internet service provider. An application should always check the return value in case there is an error; never assume that the return value is always a valid address. The function may be unable to determine the external IP address for the local host for a number of reasons, particularly if the system is behind a firewall or uses a proxy server that restricts access to external sites on the Internet. If the function is able to obtain a valid external address for the local host, that address will be cached by the library for sixty minutes. Because dial-up connections typically have different IP addresses assigned to them each time the system is connected to the Internet, it is

recommended that this function only be used with broadband connections where a NAT router is being used.

Calling this function may cause the current thread to block until the external IP address can be resolved and should never be used in conjunction with asynchronous socket connections. If you need to call this function in an application which uses asynchronous sockets, it is recommended that you create a new thread and call this function from within that thread.

To convert the address from its binary form into a string, use the **InetFormatAddress** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cssock10.lib`

See Also

[InetFormatAddress](#), [InetGetHostAddress](#), [InetGetLocalAddress](#), [InetGetPeerAddress](#)

InetGetHostAddress Function

```
INT WINAPI InetGetHostAddress(  
    LPCTSTR lpszHostName,  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddress  
);
```

The **InetGetHostAddress** function resolves the specified host name into an IP address in binary format.

Parameters

lpszHostName

A pointer to the name of the host to resolve; this may be a fully-qualified domain name or an IP address. This function recognizes the format for both IPv4 and IPv6 format addresses.

nAddressFamily

An integer which identifies the type of IP address to return. It may be one of the following values:

Constant	Description
INET_ADDRESS_UNKNOWN	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on how the host name can be resolved. By default, a preference will be given for returning an IPv4 address. However, if the host only has an IPv6 address, that value will be returned.
INET_ADDRESS_IPV4	Specifies that the address should be returned in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address should be returned in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the IP address of the specified host.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

This function can also be used to convert an address in dot notation to a binary format. If the function must perform a DNS lookup to resolve the hostname, the calling thread will block. To ensure future compatibility with IPv6 networks, it is important that the application does not make

any assumptions about the format of the address. If the function returns successfully, the *ipFamily* member of the **INTERNET_ADDRESS** structure should always be checked to determine the type of address.

The *nAddressFamily* parameter is used to specify a preference for the type of address returned, however it is possible that a host may not have an IPv4 or IPv6 address record, in which case this function will fail. Although IPv4 is still the most common address used at this time, an application should not assume that because a given host name does not have an IPv4 address, that the host name is invalid.

If the *nAddressFamily* parameter is specified as `INET_ADDRESS_UNKNOWN`, the application must be prepared to handle IPv6 addresses because it is possible for a host name to have an IPv6 address assigned to it and no IPv4 address. For legacy applications that only recognize IPv4 addresses, the *nAddressFamily* member should always be specified as `INET_ADDRESS_IPV4` to ensure that only IPv4 addresses are returned.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **InetIsProtocolAvailable** function. If an IPv6 stack is not installed, this function will fail if the *lpszHostName* parameter specifies an host that only has an IPv6 (AAAA) DNS record.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetGetHostName](#), [InetGetLocalAddress](#), [InetGetLocalName](#), [InetGetPeerAddress](#), [InetIsProtocolAvailable](#), [INTERNET_ADDRESS](#)

InetGetHostFile Function

```
INT WINAPI InetGetHostFile(  
    LPTSTR lpszFileName,  
    INT nMaxLength  
);
```

The **InetGetHostFile** function returns the name of the host file previously set using the **InetSetHostFile** function. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **InetGetHostAddress** and **InetGetHostNames** function.

Parameters

lpszFileName

Pointer to a string buffer that will contain the host file name. It is recommended that this buffer be at least MAX_PATH characters in size. This parameter may be NULL, in which case the function will return the length of the string, not including the terminating null character.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the function succeeds, the return value is length of the string. A return value of zero indicates that no host file has been specified or the function was unable to determine the file name. To get extended error information, call **InetGetLastError**. If the last error is zero, this indicates that no host file name has been specified for the current thread. If the last error is non-zero, this indicates the reason that the function failed.

Remarks

This function only returns the name of the host file that is cached in memory for the current thread. The contents of the file on the disk may have changed after the file was loaded into memory. To reload the host file or clear the cache, call the **InetSetHostFile** function.

If a host file has been specified, it is processed before the default host file when resolving a hostname into an IP address, or an IP address into a hostname. If the host name or address is not found, or no host file has been specified, a nameserver lookup is performed.

The host file returned by this function may be different than the default host file for the local system. To determine the file name for the default host file, use the **InetGetDefaultHostFile** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetGetDefaultHostFile](#), [InetGetHostAddress](#), [InetGetHostName](#), [InetSetHostFile](#)

InetGetHostName Function

```
INT WINAPI InetGetHostName(  
    LPINTERNET_ADDRESS LpAddress,  
    LPTSTR LpszHostName,  
    INT cchHostName  
);
```

The **InetGetHostName** function performs a reverse lookup, returning the host name associated with a given IP address.

Parameters

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure which specifies the IP address that should be resolved into a host name.

lpszHostName

A pointer to the buffer that will contain the host name. It is recommended that this buffer be at least 256 characters in length to accommodate the longest possible fully qualified domain name.

cchHostName

The maximum number of characters that can be copied into the buffer.

Return Value

If the function succeeds, the return value is the length of the hostname. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

If the function must perform a reverse DNS lookup to resolve the IP address into a host name, the calling thread will block. This function requires that the host have a PTR record, otherwise it will fail. Because many hosts do not have a PTR record, calling this function frequently may have a negative impact on the overall performance of the application.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **InetIsProtocolAvailable** function. If an IPv6 stack is not installed, this function will fail if the *lpAddress* parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetGetHostAddress](#), [InetGetLocalAddress](#), [InetGetLocalName](#), [InetGetPeerAddress](#), [InetIsProtocolAvailable](#), [INTERNET_ADDRESS](#)

InetGetLastError Function

```
DWORD WINAPI InetGetLastError();
```

Parameters

None.

Return Value

Functions set this value by calling the [InetSetLastError](#) function. The return value section of each reference page notes the conditions under which the function sets the last-error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **InetGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **InetSetLastError(0)** when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SOCKET or INET_ERROR. Those functions which call **InetSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[InetGetErrorString](#), [InetSetLastError](#)

InetGetLocalAddress Function

```
INT WINAPI InetGetLocalAddress(  
    SOCKET hSocket,  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS LpAddress,  
    UINT* LpnPort  
);
```

The **InetGetLocalAddress** function returns the local IP address and port number for the specified socket.

Parameters

hSocket

The socket handle.

nAddressFamily

An integer which identifies the type of IP address to return. It may be one of the following values:

Constant	Description
INET_ADDRESS_UNKNOWN	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on the type of connection that was established. If the <i>hSocket</i> parameter is INVALID_SOCKET, a preference will be given for returning an IPv4 address. However, if the local host only has an IPv6 address, that value will be returned.
INET_ADDRESS_IPV4	Specifies that the address should be returned in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address should be returned in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the IP address of the local host. If the *hSocket* parameter is specified as INVALID_SOCKET, this function will attempt to determine the IP address of the local host assigned by the system. If the address is not required, this parameter may be NULL.

lpnPort

A pointer to an unsigned integer that will contain the local port number. If the *hSocket* parameter specifies a valid socket, this parameter will be set to the local port that the socket was bound to. If the *hSocket* parameter is specified as INVALID_SOCKET, this parameter is ignored. If the port number is not required, this parameter may be NULL.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

To ensure future compatibility with IPv6 networks, it is important that the application does not make any assumptions about the format of the address. If the function returns successfully, the *ipFamily* member of the **INTERNET_ADDRESS** structure should always be checked to determine the type of address.

If the *nAddressFamily* parameter is specified as `INET_ADDRESS_UNKNOWN`, the application must be prepared to handle IPv6 addresses because it is possible for the local host to have an IPv6 address assigned to it and no IPv4 address. For legacy applications that only recognize IPv4 addresses, the *nAddressFamily* member should always be specified as `INET_ADDRESS_IPV4` to ensure that only IPv4 addresses are returned.

If the system is connected to the Internet through a local network and/or uses a router that performs Network Address Translation (NAT), the **InetGetLocalAddress** function will return the local, non-routable IP address assigned to the system. To determine the public IP address has been assigned to the system, you should use the **InetGetExternalAddress** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetGetExternalAddress](#), [InetGetHostAddress](#), [InetGetHostName](#), [InetGetLocalName](#), [InetGetPeerAddress](#), [INTERNET_ADDRESS](#)

InetGetLocalName Function

```
INT WINAPI InetGetLocalName(  
    LPTSTR LpszHostName,  
    INT cchHostName  
);
```

The **InetGetLocalName** function returns the hostname assigned to the local system.

Parameters

lpszHostName

A pointer to the buffer that will contain the hostname.

cchHostName

The maximum number of characters that can be copied into the address buffer.

Return Value

If the function succeeds, the return value is the length of the hostname. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetGetHostAddress](#), [InetGetHostName](#), [InetGetLocalAddress](#), [InetGetPeerAddress](#)

InetGetLockedServer Function

```
SOCKET WINAPI InetGetLockedServer(  
    LPDWORD LpdwThreadId  
);
```

The **InetGetLockedServer** function unlocks the specified server, allowing other server threads to resume execution.

Parameters

LpdwThreadId

A pointer to an unsigned integer which identifies the thread that established the server lock. If this information is not required, this parameter may be NULL.

Return Value

If the function succeeds, the return value is the handle to the locked server. If no server is in a locked state, the function will return a value of INVALID_SOCKET.

Remarks

The **InetGetLockedServer** function can be used to determine if there is a server in a locked state. If there is, the function will return a handle to the server and will identify the thread which established the lock. This function may be called from any thread, however only the thread which established the server lock may interact with the server or release the lock.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetServerLock](#), [InetServerUnlock](#)

InetGetOption Function

```
INT WINAPI InetGetOption(  
    SOCKET hSocket,  
    DWORD dwOption,  
    LPBOOL lpbEnabled  
);
```

The **InetGetOption** function is used to determine if a specific socket option has been enabled.

Parameters

hSocket

The socket handle.

dwOption

An unsigned integer used to specify one of the socket options. These options cannot be combined. The following values are recognized:

Constant	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_KEEPAIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option is commonly used by server applications.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.

lpbEnabled

A pointer to a boolean flag. If the option is enabled, the flag is set to a non-zero value, otherwise it is set to a value of zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetAsyncConnectEx](#), [InetConnectEx](#), [InetSetOption](#)

InetGetPeerAddress Function

```
INT WINAPI InetGetPeerAddress(  
    SOCKET hSocket,  
    LPINTERNET_ADDRESS lpAddress,  
    UINT* lpnPort  
);
```

The **InetGetPeerAddress** function returns the peer IP address and remote port number for the specified socket.

Parameters

hSocket

The socket handle.

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the IP address of the remote host that the socket is connected to.

lpnPort

A pointer to an unsigned integer that will contain the remote port that the socket is connected to.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

If this function is called by a server application in response to a INET_EVENT_ACCEPT event, it will return the IP address and port number for the client that is attempting to establish the connection. If the peer address is unavailable, the *ipFamily* member of the INTERNET_ADDRESS structure will be zero. To convert the IP address to a printable string, use the **InetFormatAddress** function.

It is not recommended that you use the port number for anything other than informational and logging purposes. Server applications should not make any assumptions about the specific port number or range of port numbers that a client is using when establishing a connection to the server. The ephemeral port number that a client is bound to can vary based on the client operating system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[InetFormatAddress](#), [InetGetHostAddress](#), [InetGetHostName](#), [InetGetLocalAddress](#), [InetGetLocalName](#), [INTERNET_ADDRESS](#)

InetGetPhysicalAddress Function

```
BOOL WINAPI InetGetPhysicalAddress(  
    LPTSTR lpszAddress,  
    UINT nMaxLength  
);
```

Return the media access control (MAC) address for the primary network adapter.

Parameters

lpszAddress

A string buffer that will contain the address in a printable format when the function returns. This parameter cannot be NULL.

nMaxLength

The maximum number of characters that can be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetPhysicalAddress** function returns the media access control (MAC) address for the primary network adapter. This is a 48 bit or 64 bit address that is assigned to each network interface and is used for identification and access control. All network devices on the same subnet must be assigned their own unique MAC address. Unlike IP addresses which may be assigned dynamically and can be frequently changed, MAC addresses are considered to be more permanent because they are usually assigned by the device manufacturer and stored in firmware. Note that in some cases it is possible to change the address assigned to a device, and virtual network interfaces may have configurable MAC addresses.

This function returns the MAC address as a printable string, with each byte of the address as a two-digit hexadecimal value separated by a colon. The string buffer passed to the function should be at least 20 characters long to accommodate the address and terminating null character. An example of a 48 bit address would be "01:23:45:67:89:AB". If the local system is multi-homed (having more than one network adapter) then this function will return the MAC address for the primary network adapter.

This function is provided for backwards compatibility with previous versions of the library and it is recommended that new applications use the **InetGetAdapterAddress** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetEnumNetworkAddresses](#), [InetGetAdapterAddress](#), [InetGetLocalName](#), [InetGetHostAddress](#)

InetGetSecurityInformation Function

```
BOOL WINAPI InetGetSecurityInformation(  
    SOCKET hSocket,  
    LPSECURITYINFO lpSecurityInfo  
);
```

The **InetGetSecurityInformation** function fills a structure with information about the security characteristics of a connection.

Parameters

hSocket

Handle to the socket.

lpSecurityInfo

A pointer to a [SECURITYINFO](#) structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

This function is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value `SECURITY_PROTOCOL_NONE`.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;  
  
securityInfo.dwSize = sizeof(SECURITYINFO);  
if (InetGetSecurityInformation(hClient, &securityInfo))  
{  
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)  
    {  
        MessageBox(NULL, _T("The connection is not secure"),  
                    _T("Connection"), MB_OK);  
    }  
    else  
    {  
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)  
        {  
            MessageBox(NULL, _T("The connection is secure"),  
                        _T("Connection"), MB_OK);  
        }  
        else  
        {  
            MessageBox(NULL, _T("The server certificate not valid"),  
                        _T("Connection"), MB_OK);  
        }  
    }  
}
```

```
}  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: csoskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetConnectEx](#), [InetAsyncConnectEx](#)

InetGetServerClient Function

```
SOCKET WINAPI InetGetServerClient(  
    SOCKET hServer  
);
```

The **InetGetServerClient** function returns the handle for the last client connection accepted by the server.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is the handle to the last client connection that was accepted by the server. If the function fails, the return value is `INVALID_SOCKET`. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetServerClient** function can be used inside the service event handler to determine the client connection that was just accepted by the server. This would typically be used in conjunction with the `INET_EVENT_ACCEPT` handler, enabling the application to obtain the handle of the new client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetGetClientThreadId](#), [InetGetServerThreadId](#), [InetServerBroadcast](#), [InetServerLock](#), [InetServerStop](#), [InetServerThrottle](#), [InetServerUnlock](#)

InetGetServerData Function

```
BOOL WINAPI InetGetServerData(  
    SOCKET hServer,  
    LPVOID * lppvData  
);
```

The **InetGetServerData** function returns the application defined data associated with the specified server.

Parameters

hSocket

The server socket handle.

lppvData

Pointer to a void pointer which will contain an application defined value associated with the server.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates that application defined data for the server could not be retrieved. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetServerData** function is used to retrieve the application defined data that was previously associated with a server using the **InetSetServerData** function. This is typically used to associate a pointer to a data structure or a class instance with a specific instance of a server.

This function can only be used with socket handles created using **InetServerStart** function. It cannot be used with socket handles created using the **InetListen** or **InetListenEx** functions. If the socket handle is invalid, or does not reference a server handle, the *lppvData* pointer passed to this function will be initialized to a value of NULL and the function will return a value of zero.

If this function is called with a valid socket handle and there is no data associated with the socket, the function will return a non-zero value and the *lppvData* pointer will be returned with a NULL value. Before dereferencing the pointer returned by this function, the application should always check the return value to ensure the function succeeded and make sure that the pointer is not NULL.

Example

```
UINT *pnValue1 = (UINT *)LocalAlloc(LPTR, sizeof(UINT));  
UINT *pnValue2 = NULL;  
  
*pnValue1 = 1234;  
  
if (InetSetServerData(hServer, pnValue1) == FALSE)  
{  
    // Unable to associate the data with this server  
    return;  
}  
  
if (InetGetServerData(hServer, &pnValue2) == FALSE)  
{  
    // Unable to retrieve the data associated with this server
```

```
    return;  
}  
  
// *pnValue2 == 1234  
printf("The value of user defined data is %u\n", *pnValue2);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetGetClientData](#), [InetSetClientData](#), [InetSetServerData](#)

InetGetServerPriority Function

```
INT WINAPI InetGetServerPriority(  
    SOCKET hServer  
);
```

The **InetGetServerPriority** function returns the current priority for the specified server.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is the priority for the specified server. If the function fails, the return value is `INET_PRIORITY_INVALID`. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetServerPriority** function can be used to determine the current priority assigned to the specified server. It will be one of the following values:

Constant	Description
<code>INET_PRIORITY_BACKGROUND</code>	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
<code>INET_PRIORITY_LOW</code>	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
<code>INET_PRIORITY_NORMAL</code>	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
<code>INET_PRIORITY_HIGH</code>	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
<code>INET_PRIORITY_CRITICAL</code>	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

The socket handle for the server must be one that was created using the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetGetClientPriority](#), [InetServerStart](#), [InetSetClientPriority](#), [InetSetServerPriority](#)

InetGetServerStackSize Function

```
DWORD WINAPI InetGetServerStackSize(  
    SOCKET hServer  
);
```

Return the initial size of the stack allocated for threads created by the server.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is the amount of memory that will be allocated for the stack in bytes. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetServerStackSize** function returns the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetServerStart](#), [InetSetServerStackSize](#)

InetGetServerStatus Function

```
INT WINAPI InetGetServerStatus(  
    SOCKET hServer  
);
```

The **InetGetServerStatus** function returns the current status of the specified server.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is the status for the specified server. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetServerStatus** function can be used to determine the current status for the specified server. It may be one of the following values:

Constant	Description
INET_SERVER_INACTIVE	The server is currently inactive.
INET_SERVER_STARTED	The server has initialized and is preparing to listen for client connections.
INET_SERVER_LISTENING	The server is actively listening for incoming client connections.
INET_SERVER_SUSPENDED	The server has been suspended and is no longer accepting client connections.
INET_SERVER_SHUTDOWN	The server has been stopped and is in the process of terminating all active client connections.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetServerRestart](#), [InetServerResume](#), [InetServerStart](#), [InetServerStop](#), [InetServerSuspend](#)

InetGetServerThreadId Function

```
DWORD WINAPI InetGetServerThreadId(  
    SOCKET hServer  
);
```

The **InetGetServerThreadId** function returns the thread ID for the specified server.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is an unsigned integer value which identifies the thread that was created to manage the server. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The thread ID returned by this function can be used with the **OpenThread** function to obtain a handle to the thread.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetGetClientPriority](#), [InetGetClientThreadId](#), [InetGetThreadClient](#), [InetSetClientPriority](#)

InetGetServiceName Function

```
BOOL WINAPI InetGetServiceName(  
    UINT nServicePort,  
    LPTSTR lpszServiceName,  
    INT nMaxLength  
);
```

The **InetGetServiceName** function returns the service name associated with a specified port number.

Parameters

nServicePort

Port number associated with some network service.

lpszServiceName

A pointer to a string buffer that will contain the name of the service associated with the specified port number. This string should be at least 32 characters long.

cchServiceName

An integer value which specifies the maximum number of characters that can be copied into the string buffer.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetGetServicePort](#)

InetGetServicePort Function

```
UINT WINAPI InetGetServicePort(  
    LPCTSTR lpszServiceName  
);
```

The **InetGetServicePort** function returns the port number associated with a service name.

Parameters

lpszServiceName

A pointer to a string which specifies the name of the service to return the port number for.

Return Value

If the function succeeds, the return value is the port number associated with a service name. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetGetServiceName](#)

InetGetStatus Function

```
INT WINAPI InetGetStatus(  
    SOCKET hSocket  
);
```

The **InetGetStatus** function is used to report what sort of socket operation is in progress.

Parameters

hSocket

The socket handle.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

The return value is one of the following values:

Value	Constant	Description
0	<code>INET_STATUS_UNUSED</code>	No connection has been established.
1	<code>INET_STATUS_IDLE</code>	The socket is idle and not in a blocked state
2	<code>INET_STATUS_LISTEN</code>	The socket is listening for inbound connections from a client
3	<code>INET_STATUS_CONNECT</code>	The socket is establishing a connection with a server
4	<code>INET_STATUS_ACCEPT</code>	The socket is accepting a connection from a client
5	<code>INET_STATUS_READ</code>	Data is being read from the socket
6	<code>INET_STATUS_WRITE</code>	Data is being written to the socket
7	<code>INET_STATUS_FLUSH</code>	The socket is being flushed; all data in the receive buffers is being discarded
8	<code>INET_STATUS_DISCONNECT</code>	The socket is disconnecting from the remote host

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified socket.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetGetBlockingSocket](#), [InetIsConnected](#), [InetIsListening](#), [InetIsReadable](#), [InetIsWritable](#)

InetGetStreamInfo Function

```
BOOL WINAPI InetGetStreamInfo(  
    SOCKET hSocket,  
    LPINETSTREAMINFO lpStreamInfo  
);
```

The **InetGetStreamInfo** function fills a structure with information about the current stream I/O operation.

Parameters

hSocket

Handle to the socket.

lpSecurityInfo

A pointer to an **INETSTREAMINFO** structure which contains information about the status of the current operation.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetStreamInfo** function returns information about the current streaming socket operation, including the average number of bytes transferred per second and the estimated amount of time until the operation completes. If there is no operation currently in progress, this function will return the status of the last successful streaming read or write performed by the client.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified socket.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetReadStream](#), [InetStoreStream](#), [InetWriteStream](#), [INETSTREAMINFO](#)

InetGetThreadClient Function

```
SOCKET WINAPI InetGetThreadClient(  
    DWORD dwThreadId  
);
```

The **InetGetThreadClient** function returns the socket handle for the client session that is being managed by the specified thread.

Parameters

dwThreadId

An unsigned integer value which identifies the thread managing the client session. If this parameter has a value of zero, then the client handle for the current thread is returned.

Return Value

If the function succeeds, the return value is the socket handle for the specified client session. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetThreadClient** is used to obtain the socket handle for the client session that is being managed by the specified thread. If the specified thread ID is zero, then the function will return the client socket for the current thread, otherwise it will search the internal table of all active client sessions and return the handle to the session that is being managed by that thread.

This function will fail if the thread ID does not specify an active client session thread.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetGetClientHandle](#), [InetGetClientId](#), [InetGetClientThreadId](#), [InetGetServerClient](#),
[InetGetServerThreadId](#)

InetGetTimeout Function

```
INT WINAPI InetGetTimeout(  
    SOCKET hSocket  
);
```

The **InetGetTimeout** function returns the timeout interval for blocking operations, in seconds.

Parameters

hSocket

Handle to the socket.

Return Value

If the function succeeds, the return value is the timeout interval for blocking operations, in seconds. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetSetTimeout](#)

InetHostNameToUnicode Function

```
INT WINAPI InetHostNameToUnicode(  
    LPCTSTR LpszHostName,  
    LPTSTR LpszUnicodeName,  
    INT nMaxLength  
);
```

The **InetHostNameToUnicode** function converts the canonical form of a host name to its Unicode version.

Parameters

LpszHostName

Pointer to the host name as a null-terminated string. This parameter cannot be a NULL pointer or a zero length string.

LpszUnicodeName

Pointer to the string buffer that will contain the original Unicode version of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer.

nMaxLength

The maximum number of characters that can be copied to the *LpszUnicodeName* string buffer. This parameter cannot be zero, and must include the terminating null character.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer. If the function fails, the return value is INET_ERROR. To get extended error information, call **DnsGetLastError**.

Remarks

The **InetHostNameToUnicode** function will convert the encoded ASCII version of a host name to its Unicode version. Although any valid host name is accepted by this function, it is intended to convert a Punycode encoded host name to its original Unicode character encoding.

If the Unicode version of this function is used, the value returned in *LpszUnicodeName* will be a Unicode string using UTF-16 encoding. If the ANSI version of this function, the value returned will be a Unicode string using UTF-8 encoding. To display a UTF-8 encoded host name, your application will need to convert it to UTF-16 using the **MultiByteToWideChar** function.

Although this function performs checks to ensure that the *LpszHostName* parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the existence of the domain name. To check if the host name exists and has a valid IP address, use the [InetGetHostAddress](#) function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

InetInitialize Function

```
BOOL WINAPI InetInitialize(  
    LPCTSTR lpszLicenseKey,  
    LPINITDATA lpData  
);
```

The **InetInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpData

Pointer to an INITDATA data structure. This parameter may be NULL if the initialization data for the library is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**. All other functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must insure that this function is called with a valid license key.

If the *lpData* argument is specified, it must point to an INITDATA structure which has been initialized by setting the *dwSize* member to the size of the structure. All other structure members should be set to zero. If the function is successful, then the INITDATA structure will be filled with identifying information about the library.

Although it is only required that **InetInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **InetUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **InetInitialize** or **InetUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

InetIsAddressNull Function

```
BOOL WINAPI InetIsAddressNull(  
    LPINTERNET_ADDRESS lpAddress  
);
```

The **InetIsAddressNull** function determines if the IP address is null.

Parameters

lpAddress

A pointer to an INTERNET_ADDRESS structure that contains the address to check.

Return Value

If the function succeeds and the IP address is null, or the *lpAddress* parameter is a NULL pointer, the return value is non-zero. If the function fails or the address is not null, the return value is zero. If the address family is not supported, the last error code will be updated. If the address is valid but not null, the last error code will be set to NO_ERROR.

Remarks

A null IP address is one where all bits for the address (32 bits for IPv4 or 128 bits for IPv6) are zero. This is a special address that is typically used when creating a passive socket that should listen for connections on all available network interfaces.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetGetAddress](#), [InetIsAddressRoutable](#), [INTERNET_ADDRESS](#)

InetIsAddressRoutable Function

```
BOOL WINAPI InetIsAddressRoutable(  
    LPINTERNET_ADDRESS lpAddress  
);
```

The **InetIsAddressRoutable** function determines if the IP address is routable over the Internet.

Parameters

lpAddress

A pointer to an INTERNET_ADDRESS structure that contains the address to check. This parameter cannot be NULL.

Return Value

If the function succeeds and the IP address is routable over the Internet, the return value is non-zero. If the function fails or the address is not routable, the return value is zero. If the parameter is NULL, or the address family is not supported, the last error code will be updated. If the address is valid but not routable, the last error code will be set to NO_ERROR.

Remarks

A routable IP address is one that can be reached by anyone over the public Internet. These are also commonly referred to as "public addresses" which are typically assigned to networks and individual hosts by an Internet service provider. There are also certain addresses that are not routable over the Internet, and used to address systems over a local network or private intranet. This function can be used to determine if a given IP address is public (routable) or private.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetGetAddress](#), [InetGetExternalAddress](#), [InetIsAddressNull](#), [INTERNET_ADDRESS](#)

InetIsBlocking Function

```
BOOL WINAPI InetIsBlocking(  
    SOCKET hSocket  
);
```

The **InetIsBlocking** function is used to determine if the socket is performing a blocking operation.

Parameters

hSocket

Socket handle.

Return Value

If the socket is currently performing a blocking operation, the function returns a non-zero value. If the socket is not performing a blocking operation, or the socket handle is invalid, the function returns zero.

Remarks

This function is typically used to determine if an open socket that is being used by another thread is currently blocked. A socket may block when waiting to receive data from a remote host or while data is actively being exchanged. Because there can only be one blocking socket operation per thread, this function can be used to determine if a function such as **InetRead** or **InetWrite** would fail because another thread is currently sending or receiving data on that socket. This socket handle that is passed to this function does not need to be owned by the current thread.

It is important to note that if this function returns a non-zero value, it does not guarantee that a subsequent read or write on the socket will succeed. The application should always check the return value from functions such as **InetRead** and **InetWrite** to ensure they were successful.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetIsConnected](#), [InetIsReadable](#), [InetIsWritable](#), [InetRead](#), [InetWrite](#)

InetIsClosed Function

```
BOOL WINAPI InetIsClosed(  
    SOCKET hSocket  
);
```

The **InetIsClosed** function is used to determine if the remote host has closed its socket.

Parameters

hSocket

Socket handle.

Return Value

If the remote host has closed its socket, the function returns a non-zero value. If the remote host has not closed its connection, or the socket handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetIsConnected](#), [InetIsListening](#), [InetIsReadable](#), [InetIsWritable](#)

InetIsConnected Function

```
BOOL WINAPI InetIsConnected(  
    SOCKET hSocket  
);
```

The **InetIsConnected** function is used to determine if the socket is currently connected to a remote host.

Parameters

hSocket

Socket handle.

Return Value

If the socket is connected to a remote host, the function returns a non-zero value. If the socket is not connected, or the socket handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetIsClosed](#), [InetIsListening](#), [InetIsReadable](#), [InetIsWritable](#)

InetIsListening Function

```
BOOL WINAPI InetIsListening(  
    SOCKET hSocket  
);
```

The **InetIsListening** function determines if the socket is listening for connection requests.

Parameters

hSocket

Socket handle.

Return Value

If the socket is being used to listen for connection requests, the function returns a non-zero value. If the socket is not listening or the socket handle is invalid, the function returns zero.

Remarks

The **InetIsListening** function determines if the socket is being used in a server application to actively listen for incoming connection requests from client applications. A listening socket can be created using either the **InetAsyncListen** or **InetListen** functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetAsyncListen](#), [InetIsReadable](#), [InetIsWritable](#), [InetIsConnected](#), [InetListen](#)

InetIsProtocolAvailable Function

```
BOOL WINAPI InetIsProtocolAvailable(  
    INT nAddressFamily,  
    INT nProtocol  
);
```

The **InetIsProtocolAvailable** function determines if the operating system supports creating a socket for the specified address family and protocol.

Parameters

nAddressFamily

An integer which identifies the address family that should be checked. It should be one of the following values:

Constant	Description
INET_ADDRESS_IPV4	Specifies that the function should determine if it can create an Internet Protocol version 4 (IPv4) socket. This requires that the system have an IPv4 TCP/IP stack bound to at least one network adapter on the local system. All Windows systems include support for IPv4 by default.
INET_ADDRESS_IPV6	Specifies that the function should determine if it can create an Internet Protocol version 6 (IPv6) socket. This requires that the system have an IPv6 TCP/IP stack bound to at least one network adapter on the local system. Windows XP and Windows Server 2003 includes support for IPv6, however it is not installed by default. Windows Vista and later versions include support for IPv6 and enable it by default.

nProtocol

An integer which identifies the protocol that should be checked. It should be one of the following values:

Constant	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This requires that the system be capable of creating a stream socket using the specified address family.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. This requires that the system be capable of creating a datagram socket using the specified address family.

Return Value

If the the system is capable of creating a socket using the specified address family and protocol, this function will return a non-zero value. If the combination of address family and protocol is not supported, this function will return a value of zero.

Remarks

The **InetIsProtocolAvailable** function is used to determine if the operating system supports creating a particular type of socket. Typically it is used by an application to determine if the system has an IPv6 TCP/IP stack installed and configured. By default, all Windows systems will have an IPv4 stack installed if the system has a network adapter. However, not all systems may have an IPv6 stack installed, particularly older Windows XP and Windows Server 2003 systems. Note that if an IPv6 stack is not installed, the library will not recognize IPv6 addresses and cannot resolve host names that only have an IPv6 (AAAA) record, even if the address or host name is valid.

Example

```
if (!InetIsProtocolAvailable(INET_ADDRESS_IPV6, INET_PROTOCOL_TCP))
{
    AfxMessageBox(_T("This system does not support IPv6"), MB_ICONEXCLAMATION);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetAddress](#), [InetGetHostAddress](#), [InetGetHostName](#)

InetIsReadable Function

```
BOOL WINAPI InetIsReadable(  
    SOCKET hSocket,  
    DWORD dwTimeout,  
    LPDWORD lpdwAvail  
);
```

The **InetIsReadable** function is used to determine if data is available to be read from the socket.

Parameters

hSocket

Socket handle.

dwTimeout

Timeout value in milliseconds. If the socket cannot be read within this time period, the function will return a value of zero. A timeout value of zero specifies that the socket should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read.

Return Value

If the current thread can read data from the socket without blocking, the function returns a non-zero value. If the current thread cannot read any data without blocking, the function returns zero.

Remarks

On some platforms, the value returned in *lpdwAvail* will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

If the connection is secure, the value returned in *lpdwAvail* will reflect the number of bytes available in the encrypted data stream. The actual amount of data available to the application after it has been decrypted will vary.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetIsClosed](#), [InetIsWritable](#), [InetPeek](#), [InetRead](#), [InetReadLine](#), [InetReadStream](#)

InetIsUrgent Function

```
BOOL WINAPI InetIsUrgent(  
    SOCKET hSocket  
);
```

The **InetIsUrgent** function determines if there is any out-of-band (OOB) data available to be read.

Parameters

hSocket

Handle to the socket.

Return Value

If there is out-of-band data, the return value is non-zero. If there is no out-of-band data, or an error occurs the return value is zero. To determine if an error has occurred, use the **InetGetLastError** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetSetOption \(INET_OPTION_INLINE\)](#)

InetIsWritable Function

```
BOOL WINAPI InetIsWritable(  
    SOCKET hSocket,  
    DWORD dwTimeout  
);
```

The **InetIsWritable** function is used to determine if data can be written to the socket.

Parameters

hSocket

Socket handle.

dwTimeout

Timeout value in milliseconds. If the socket cannot be written to within this time period, the function will return a value of zero. A timeout value of zero specifies that the socket should be polled without blocking the current thread.

Return Value

If the current thread can write data to the socket within the timeout period, the function returns a non-zero value. The function will return zero if the socket send buffer is full.

Remarks

The **InetIsWritable** function cannot be used to determine the amount of data that can be sent to the remote host without blocking the current thread. A non-zero return value only indicates that the send buffer is not full and can accept some data. In most cases, it is recommended that larger blocks of data be broken into smaller logical blocks rather than attempting to send it all of the data at once. For very large streams of data, it is recommended that you use the **InetWriteStream** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

See Also

[InetIsReadable](#), [InetWrite](#), [InetWriteLine](#), [InetWriteStream](#)

InetListen Function

```
SOCKET WINAPI InetListen(  
    LPCTSTR lpszLocalAddress,  
    UINT nLocalPort  
);
```

The **InetListen** function creates a passive socket used to listen for connections from a client application.

This function is included for backwards compatibility with legacy applications. New projects should use the **InetServerStart** function to create a server application.

Parameters

lpszLocalAddress

A pointer to a string which specifies the local IP address that the socket should be bound to. If this parameter is NULL or points to an empty string, a client may establish a connection using any valid network interface configured on the local system. If an address is specified, then a client may only establish a connection with the system using that address.

nLocalPort

The local port number that the socket should be bound to. This value must be greater than zero. Port numbers less than 1024 are considered reserved ports and may require that the process execute with administrative privileges and/or require changes to the default firewall rules to permit inbound connections.

Return Value

If the function succeeds, the return value is a socket handle. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

The socket option INET_OPTION_REUSEADDRESS is enabled by default when calling the **InetListen** function. This allows an application to re-use a local address and port number when creating the listening socket. If this behavior is not desired, use the **InetListenEx** function instead.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the function will fail.

After the listening socket has been created, the application should then call the **InetAccept** function to wait for a client to establish a connection. For servers that need to handle multiple simultaneous client connections, it is recommended that the asynchronous functions be used.

Example

```
SOCKET hServer = INVALID_SOCKET;  
LPCTSTR lpszAddress = _T("192.168.0.48");  
  
// Accept connections from clients that connect to
```

```
// address 192.168.0.48 on port 7000

hServer = InetListen(lpszAddress, 7000);
if (hServer == INVALID_SOCKET)
{
    DWORD dwError;
    TCHAR szError[256];

    dwError = InetGetLastError();
    InetGetErrorString(dwError, szError, 256);

    MessageBox(NULL, szError, NULL, MB_OK|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cswsock10.h

Import Library: cswskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetAccept](#), [InetAddress](#), [InetEnableEvents](#), [InetInitialize](#), [InetListenEx](#), [InetReject](#), [InetServerStart](#)

InetListenEx Function

```
SOCKET WINAPI InetListenEx(  
    LPCTSTR lpszLocalAddress,  
    UINT nLocalPort,  
    UINT nBackLog,  
    DWORD dwOptions  
);
```

The **InetListenEx** function creates a passive socket, and specifies the maximum number of connection requests that will be queued.

Parameters

lpszLocalAddress

A pointer to a string which specifies the local IP address that the socket should be bound to. If this parameter is NULL or points to an empty string, a client may establish a connection using any valid network interface configured on the local system. If an address is specified, then a client may only establish a connection with the system using that address.

nLocalPort

The local port number that the socket should be bound to. This value must be greater than zero. Port numbers less than 1024 are considered reserved ports and may require that the process execute with administrative privileges and/or require changes to the default firewall rules to permit inbound connections.

nBacklog

The maximum length of the queue allocated for pending client connections. A value of zero specifies that the size of the queue should be set to a maximum reasonable value. On Windows server platforms, the maximum value is large enough to queue several hundred pending connections.

dwOptions

An unsigned integer used to specify one or more socket options. The following values are supported:

Constant	Description
INET_OPTION_NONE	No option specified. If the address and port number are in use by another application or a closed socket which was listening on this port is still in the TIME_WAIT state, the function will fail.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option enables a server application to listen for connections using the specified address and port number even if they were in use recently. This is typically used to enable an application to close the listening socket and immediately reopen it without getting an error that the address is in use.
INET_OPTION_EXCLUSIVE	This option specifies the local address and port number is for the exclusive use by the current process, preventing another application from forcibly binding to the same address. If another process has

	already bound a socket to the address provided by the caller, this function will fail.
INET_OPTION_RESERVEDPORT	This option specifies the listening socket should be bound to an unused port number less than 1024, which is typically reserved for well-known system services. If this option is specified, the process may require administrative privileges and firewall rules that will permit a client to establish a connection with the service.

Return Value

If the function succeeds, the return value is a socket handle. If the function fails, the return value is `INVALID_SOCKET`. To get extended error information, call **InetGetLastError**.

Remarks

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

If the `INET_OPTION_REUSEADDRESS` option is not specified, an error may be returned if a listening socket was recently created for the same local address and port number. By default, once a listening socket is closed there is a period of time that all applications must wait before the address can be reused (this is called the `TIME_WAIT` state). The actual amount of time depends on the operating system and configuration parameters, but is typically two to four minutes. Specifying this option enables an application to immediately re-use a local address and port number that was previously in use. Note that this does not permit more than one server to bind to the same address.

If the `INET_OPTION_EXCLUSIVE` option is specified, the local address and port number cannot be used by another process until the listening socket is closed. This can prevent another application from forcibly binding to the same listening address as your server. This option can be useful in determining whether or not another process is already bound to the address you wish to use, but it may also prevent your server application from restarting immediately, regardless if the `INET_OPTION_REUSEADDRESS` option has also been specified.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the function will fail.

After the listening socket has been created, the application should then call the **InetAccept** or **InetAcceptEx** function to wait for a client to establish a connection. For servers that need to handle multiple simultaneous client connections, it is recommended that the asynchronous functions be used.

To enable asynchronous event notification, use the **InetEnableEvents** function.

Example

```
SOCKET hServer = INVALID_SOCKET;
LPCTSTR lpszAddress = _T("192.168.0.48");

// Accept connections from clients that connect to
```

```
// address 192.168.0.48 on port 7000 with a standard
// backlog of 5 connections

hServer = InetListenEx(lpszAddress,
                      7000,
                      INET_BACKLOG,
                      INET_OPTION_REUSEADDRESS);

if (hServer == INVALID_SOCKET)
{
    DWORD dwError;
    TCHAR szError[256];

    dwError = InetGetLastError();
    InetGetErrorString(dwError, szError, 256);

    MessageBox(NULL, szError, NULL, MB_OK|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetAccept](#), [InetAsyncAccept](#), [InetAsyncListen](#), [InetAsyncListenEx](#), [InetEnableEvents](#),
[InetAddress](#), [InetListen](#)

InetMatchHostName Function

```
BOOL WINAPI InetMatchHostName(  
    LPCTSTR lpszHostName,  
    LPCTSTR lpszHostMask  
    BOOL bResolve  
);
```

The **InetMatchHostName** function matches a host name against one or more strings that may contain wildcards.

Parameters

lpszHostName

A pointer to a string which specifies the host name or IP address to match.

lpszHostMask

A pointer to a string which specifies one or more values to match against the host name. The asterisk character can be used to match any number of characters in the host name, and the question mark can be used to match any single character. Multiple values may be specified by separating them with a semicolon.

bResolve

A boolean value which specifies if the host name or IP address should be resolved when matching the host against the mask string. If this parameter is non-zero, two checks against the host mask string will be performed; once for the host name specified and once for its IP address. If this parameter is zero, then the match is made only against the host name string provided.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetMatchHostName** function provides a convenient way for an application to determine if a given host name matches one or more mask strings which may contain wildcard characters. For example, the host name could be "www.microsoft.com" and the host mask string could be "*.microsoft.com". In this example, the function would return a non-zero value indicating the host name matched the mask. However, if the mask string was "*.net" then the function would return zero, indicating that there was no match. Multiple mask values can be combined by separating them with a semicolon; for example, the mask "*.com;*.org" would match any host name in either the .com or .org top-level domains.

If an internationalized domain name (IDN) is specified, it will be converted internally to an ASCII string using Punycode encoding. The host mask will be matched against this encoded version of the host name, not its Unicode version.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetAddress](#), [InetAddress.getHostAddress](#), [InetAddress.getHostByName](#), [InetAddress.getLocalAddress](#), [InetAddress.getPeerAddress](#)

InetNormalizeHostName Function

```
INT WINAPI InetNormalizeHostName(  
    LPCTSTR lpszHostName,  
    LPTSTR lpszNormalized,  
    INT nMaxLength  
);
```

The **InetNormalizeHostName** function returns the canonical form of a host name in the specified buffer.

Parameters

lpszHostName

Pointer to the host name as a null-terminated string. This parameter cannot be a NULL pointer or a zero length string.

lpszNormalized

Pointer to the string buffer that will contain the canonical form of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer.

nMaxLength

The maximum number of characters that can be copied to the *lpszNormalized* string buffer. This parameter cannot be zero, and must include the terminating null character.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetNormalizeHostName** function will remove all leading and trailing whitespace characters from the host name and fold all upper-case characters to lower-case. If an internationalized domain name (IDN) containing Unicode characters is passed to this function, it will be converted to an ASCII compatible format for domain names.

If the Unicode version of this function is used, the host name will be converted from UTF-16 to UTF-8 and then processed. If you are unsure if an internationalized domain name will be specified as the host name, it is recommended that you use the Unicode version.

Although this function performs checks to ensure that the *lpszHostName* parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the existence of the domain name. To check if the host name exists and has a valid IP address, use the [InetGetHostAddress](#) function.

It is recommended that you use this function if your application needs to store the host name, and if accepts a host name as user input. It is not necessary to call this function prior to calling the other INET API functions that accept a host name as a parameter. Those functions already normalize the host name and perform checks to ensure it is in the correct format.

If the *lpszHostName* parameter specifies a valid IPv4 or IPv6 address string instead of a host name, this function will return a copy of that IP address in the buffer provided by the caller. This allows the function to be used in cases where a user may input either a host name or IP address. To determine if the IP address has a corresponding host name, use the [InetGetHostName](#) method.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cswsock10.h

Import Library: cswskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetGetHostAddress](#), [InetGetHostName](#), [InetHostNameToUnicode](#)

InetPeek Function

```
INT WINAPI InetPeek(  
    SOCKET hSocket,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **InetPeek** function reads the specified number of bytes from the socket and copies them into the buffer, but it does not remove the data from the internal socket buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hSocket

The socket handle.

lpBuffer

Pointer to the buffer in which the data will be copied. This argument may be NULL, in which case no data is copied from the socket buffers, however the function will return the number of bytes available to read.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. If the *lpBuffer* parameter is not NULL, this value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes available to read from the socket. A return value of zero indicates that there is no data available to read at that time. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetPeek** function returns data that is available to read from the socket, up to the number of bytes specified. The data returned by this function is not removed from the socket buffers. It must be consumed by a subsequent call to the **InetRead** or **InetReadEx** function. The return value indicates the number of bytes that can be read in a single operation, up to the specified buffer size. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

If no data is available to be read, the method will return a value of zero. Using this method in a loop to poll a non-blocking socket may cause the application to become non-responsive. To determine if there is data available to be read, use the **InetIsReadable** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetFlush](#), [InetRead](#), [InetReadEx](#), [InetWrite](#), [InetWriteEx](#)

InetRead Function

```
INT WINAPI InetRead(  
    SOCKET hSocket,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **InetRead** function reads the specified number of bytes from the socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hSocket

The socket handle.

lpBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the remote host has closed the connection and there is no more data available to be read. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

The **InetRead** function will read up to the specified number of bytes and store the data in the buffer provided by the caller. If there is no data available to be read at the time this function is called, the current thread will block until at least one byte of data becomes available, the timeout period elapses or an error occurs. This function will return if any amount of data is sent by the remote host, and will not block until the entire buffer has been filled. To avoid blocking the current thread, either create an asynchronous socket or use the **InetIsReadable** function to determine if there is data available to be read prior to calling this function.

The application should never make an assumption about the amount of data that will be available to read. TCP considers all data to be an arbitrary stream of bytes and does not impose any structure on the data itself. For example, if the remote host is sending data to the server in fixed 512 byte blocks of data, it is possible that a single call to the **Read** function will return only a partial block of data, or it may return multiple blocks combined together. It is the responsibility of the application to buffer and process this data appropriately.

For applications that are built using the Unicode character set, it is important to note that the buffer is an array of bytes, not characters. If the remote host is writing string data to the socket, it must be read as a stream of bytes and converted using the **MultiByteToWideChar** function. If the remote host is sending lines of text terminated with a linefeed or carriage return and linefeed pair, the **InetReadLine** function will return a line of text at a time and perform this conversion for you.

When **InetRead** is called and the socket is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a

fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetIsReadable](#), [InetPeek](#), [InetReadEx](#), [InetWrite](#), [InetWriteEx](#)

InetReadEx Function

```
INT WINAPI InetReadEx(  
    SOCKET hSocket,  
    LPVOID LpvBuffer,  
    INT cbBuffer,  
    DWORD dwReserved,  
    LPINTERNET_ADDRESS LpRemoteAddress,  
    UINT * LpnRemotePort  
);
```

The **InetReadEx** function reads the specified number of bytes from the socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hSocket

The socket handle.

lpvBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

dwReserved

Reserved parameter. This value must always be zero.

lpRemoteAddress

Pointer to an [INTERNET_ADDRESS](#) structure which will contain the IP address of the remote host that sent the data being read. If this information is not required, the parameter may be specified as NULL.

lpnRemotePort

Pointer to an unsigned integer which will contain the remote port number. If this information is not required, the parameter may be specified as NULL.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the remote host has closed the connection and there is no more data available to be read. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

When **InetReadEx** is called and the socket is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

This function extends the **InetRead** function to return additional information about the peer who sent the data being received. For a client TCP socket, the IP address and remote port are the same values that were used to establish the connection. For a server TCP socket, it is the IP address and port number of the client which sent the data. When reading data from a UDP socket, this is the IP address and remote port of the peer that sent the datagram. This information can be used in

conjunction with the **InetWriteEx** function to send a datagram back to that host.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

See Also

[InetFlush](#), [InetGetPeerAddress](#), [InetPeek](#), [InetRead](#), [InetWrite](#), [InetWriteEx](#), [INTERNET_ADDRESS](#)

InetReadLine Function

```
BOOL WINAPI InetReadLine(  
    SOCKET hSocket,  
    LPTSTR lpszBuffer,  
    LPINT lpnLength  
);
```

The **InetReadLine** function reads up to a line of data from the socket and returns it in a string buffer.

Parameters

hSocket

The socket handle. The socket must reference a stream socket, not a datagram or raw socket. If the socket type is not valid, the function will return an error.

lpszBuffer

Pointer to the string buffer that will contain the data when the function returns. The string will be terminated with a null byte, and will not contain the end-of-line characters.

lpnLength

A pointer to an integer value which specifies the length of the buffer. The value should be initialized to the maximum number of characters that can be copied into the string buffer, including the terminating null character. When the function returns, its value will updated with the actual length of the string.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetReadLine** function reads data from the socket and copies into a specified string buffer. Unlike the **InetRead** function which reads arbitrary bytes of data, this function is specifically designed to return a single line of text data in a null-terminated string. When an end-of-line character sequence is encountered, the function will stop and return the data up to that point. The string buffer is guaranteed to be null-terminated and will not contain the end-of-line characters.

There are some limitations when using **InetReadLine**. The function should only be used to read text, never binary data. In particular, the function will discard nulls, linefeed and carriage return control characters. The Unicode version of this function will return a Unicode string, however this function does not support reading raw Unicode data from the socket. Any data read from the socket is internally buffered as octets (eight-bit bytes) and converted to Unicode using the **MultiByteToWideChar** function.

This function will force the thread to block until an end-of-line character sequence is processed, the read operation times out or the remote host closes its end of the socket connection. If this function is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, read the data and then restore the socket to asynchronous operation. If another socket operation is attempted while **InetReadLine** is blocked waiting for data from the remote host, an error will occur. It is recommended that this function only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **InetRead** and **InetReadLine** function calls can be intermixed, however be aware that **InetRead** will consume any data that has already been buffered by the **InetReadLine** function and this may have unexpected results.

Unlike the **InetRead** function, it is possible for data to be returned in the buffer even if the return value is zero. Applications should also check the value of the *lpnLength* argument to determine if any data was copied into the buffer. For example, if a timeout occurs while the function is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the function return value.

Example

```
TCHAR szBuffer[MAXBUFLEN];
INT nLength;
BOOL bResult;

do
{
    nLength = sizeof(szBuffer);
    bResult = InetReadLine(hSocket, szBuffer, &nLength);

    if (nLength > 0)
    {
        // Process the line of data returned in the string
        // buffer; the string is always null-terminated
    }
} while (bResult);

DWORD dwError = InetGetLastError();
if (dwError == ST_ERROR_CONNECTION_CLOSED)
{
    // The remote host has closed its side of the connection and
    // there is no more data available to be read
}
else if (dwError != 0)
{
    // An error has occurred while reading a line of data
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetIsReadable](#), [InetRead](#), [InetWrite](#), [InetWriteLine](#)

InetReadStream Function

```
BOOL WINAPI InetReadStream(  
    SOCKET hSocket,  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwOptions,  
    LPBYTE lpMarker,  
    DWORD cbMarker,  
    DWORD dwReserved  
);
```

The **InetReadStream** function reads the socket data stream and stores the contents in the specified buffer.

Parameters

hSocket

The socket handle. The socket must reference a stream socket, not a datagram or raw socket. If the socket type is not valid, the function will return an error.

lpvBuffer

Pointer to the buffer that will contain or reference the data when the function returns. The actual argument depends on the value of the **dwOptions** parameter which specifies how the data stream will be stored.

lpdwLength

A pointer to an unsigned integer value which specifies the maximum length of the buffer and contains the number of bytes read when the function returns. This argument should always point to an initialized value. If the *lpvBuffer* argument specifies a memory buffer, then this argument cannot point to an initialized value of zero; if any other type of stream buffer is used and the initialized value is zero, that indicates that all available data from the socket should be returned until the end-of-stream marker is encountered or the remote host disconnects.

dwOptions

An unsigned integer value which specifies both the stream buffer type and any options to be used when reading the data stream. One of the following stream types may be specified:

Constant	Description
INET_STREAM_DEFAULT	The default stream buffer type is determined by the value passed as the <i>lpvBuffer</i> parameter. If the argument specifies a pointer to a global memory handle initialized to NULL, then the function will return a handle which references the data; otherwise, the function will consider the parameter a pointer to a block of pre-allocated memory which will contain the stream data when the function returns. In most cases, it is recommended that an application explicitly specify the stream buffer type rather than using the default value.
INET_STREAM_MEMORY	The <i>lpvBuffer</i> argument specifies a pointer to a pre-allocated block of memory which will contain the data read from the socket when the function

	returns. If this stream buffer type is used, the <i>lpdwLength</i> argument must point to an unsigned integer which has been initialized with the maximum length of the buffer.
INET_STREAM_HGLOBAL	The <i>lpvBuffer</i> argument specifies a pointer to a global memory handle. When the function returns, the handle will reference a block of memory that contains the stream data. The application should take care to make sure that the handle passed to the function does not currently reference a valid block of memory; it is recommended that the handle be initialized to NULL prior to calling this function.
INET_STREAM_HANDLE	The <i>lpvBuffer</i> argument specifies a Windows handle to an open file, console or pipe. This should be the same handle value returned by the CreateFile function in the Windows API. The data read from the socket will be written to this handle using the WriteFile function.
INET_STREAM_SOCKET	The <i>lpvBuffer</i> argument specifies a socket handle. The data read from the socket specified by the <i>hSocket</i> argument will be written to this socket. The socket handle passed to this function must have been created by this library; if it is a socket created by an third-party library or directly by the Windows Sockets API, you should either attach the socket using the InetAttachSocket function or use the INET_STREAM_HANDLE stream buffer type instead.

In addition to the stream buffer types listed above, the *dwOptions* parameter may also have one or more of the following bit flags set. Programs should use a bitwise operator to combine values.

Constant	Description
INET_STREAM_CONVERT	The data stream is considered to be textual and will be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data returned in the buffer to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.

INET_STREAM_UNICODE	The data stream should be converted to Unicode. This option should only be used with text data, and will result in the stream data being returned as 16-bit wide characters rather than 8-bit bytes. The amount of data returned will be twice the amount read from the source data stream; if the application is using a pre-allocated memory buffer, this must be considered before calling this function.
---------------------	--

lpMarker

A pointer to an array of bytes which marks the end of the data stream. When this byte sequence is encountered by the function, it will stop reading and return to the caller. The buffer will contain all of the data read from the socket up to and including the end-of-stream marker. If this argument is NULL, then the function will continue to read from the socket until the maximum buffer size is reached, the remote host closes its socket or an error is encountered.

cbMarker

An unsigned integer value which specifies the length of the end-of-stream marker in bytes. If the *lpMarker* parameter is NULL, then this value must be zero.

dwReserved

A reserved parameter. This value must always be zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetReadStream** function enables an application to read an arbitrarily large stream of data and store it in memory, write it to a file or even another socket. Unlike the **InetRead** method, which will return immediately when any amount of data has been read, **InetReadStream** will only return when the buffer is full as specified by the *lpdwLength* parameter, the logical end-of-stream marker has been read, the socket closed by the remote host or when an error occurs.

This function will force the thread to block until the operation completes. If this function is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to asynchronous operation when it has finished. If another socket operation is attempted while **InetReadStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this function only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

It is possible for data to be returned in the buffer even if the function returns a value of zero. Applications should also check the value of the *lpdwLength* argument to determine if any data was copied into the buffer. For example, if a timeout occurs while the function is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the function return value.

Because **InetReadStream** can potentially cause the application to block for long periods of time as the data stream is being read, the function will periodically generate INET_EVENT_PROGRESS events. An application can register an event handler using the **InetRegisterEvent** function, and can obtain information about the current operation by calling the **InetGetStreamInfo** function.

Example

```
HGLOBAL hgblBuffer = NULL; // Return data in a global memory buffer
DWORD cbBuffer = 102400; // Read up to 100K bytes
BOOL bResult;

bResult = InetReadStream(hSocket,
                        &hgblBuffer,
                        &cbBuffer,
                        INET_STREAM_HGLOBAL | INET_STREAM_CONVERT,
                        NULL, 0, 0);

if (bResult && cbBuffer > 0)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // Use data in the stream buffer

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

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cswsock10.h

Import Library: cswskv10.lib

See Also

[InetGetStreamInfo](#), [InetRead](#), [InetReadLine](#), [InetStoreStream](#), [InetWrite](#), [InetWriteLine](#), [InetWriteStream](#)

InetRegisterEvent Function

```
INT WINAPI InetRegisterEvent(  
    SOCKET hSocket,  
    UINT nEventId,  
    INETEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

The **InetRegisterEvent** function registers a callback function for the specified event.

Parameters

hSocket

Socket handle.

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. This parameter is ignored if the socket handle specifies a server created using the **InetServerStart** function. One or more of the following values may be used:

Constant	Description
INET_EVENT_ACCEPT	A network event that indicates the process has received a connection request from a client and should accept the connection using the InetAsyncAccept function. This event is only generated for server applications which have created an asynchronous socket using the InetAsyncListen function.
INET_EVENT_CONNECT	A network event that indicates the connection to the remote host has completed.
INET_EVENT_DISCONNECT	A network event that indicates the remote host has closed the connection. The process should read any remaining data and disconnect.
INET_EVENT_READ	A network event which indicates data is available to read. No additional messages will be posted until the process has read at least some of the data from the socket. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_WRITE	A network event which indicates the application can send data to the remote host. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The application may attempt to retry the operation, or may disconnect from the remote host and report an error to the user.
INET_EVENT_CANCEL	The application has canceled a blocking operation. This

event is fired once an operation has been terminated by the InetCancel function, and control has been returned to the calling process.

lpEventProc

Specifies the address of the application defined callback function. For more information about the callback function, see the description of the **InetEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled. This parameter cannot be NULL if the socket handle specifies a server created using the **InetServerStart** function.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetRegisterEvent** function associates a callback function with a specific event. The event handler is an **InetEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

This function can be used to change the callback function and user defined parameter for a server created using the **InetServerStart** function. However, it cannot be used with client sockets automatically created by the server interface. Those sockets are managed separately in their own thread, and individual client event notifications are not supported.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[InetDisableEvents](#), [InetEnableEvents](#), [InetEventProc](#), [InetFreezeEvents](#)

InetReject Function

```
BOOL WINAPI InetReject(  
    SOCKET hSocket  
);
```

The **InetReject** function is used to reject a client connection request.

Parameters

hSocket

Handle to a listening socket.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetReject** function rejects a pending client connection and the remote host will see this as the connection being aborted. If there are no pending client connections at the time, this function will immediately return with an error indicating that the operation would cause the thread to block.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetAbort](#), [InetAccept](#), [InetListen](#)

InetServerAsyncNotify Function

```
BOOL WINAPI InetServerAsyncNotify(  
    SOCKET hServer,  
    HWND hWnd,  
    UINT uMsg  
);
```

Enable or disable asynchronous notification of changes in server status.

Parameters

hServer

The socket handle.

hWnd

A handle to the window whose window procedure will receive the notification message.

uMsg

The user-defined message that will be sent to the notification window.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerAsyncNotify** function is used by an application to enable or disable asynchronous notifications. The message window is typically the main UI window and these notifications are used signal to the application that it should update the user interface. If the *hWnd* parameter is not NULL, it must specify a valid window handle and the user-defined message must have a value of **WM_USER** or higher. The application cannot specify a notification message that is reserved by the operating system. The pseudo-handle **HWND_BROADCAST** cannot be specified as the notification window. If the *hWnd* parameter is NULL, notifications for the specified server will be disabled.

When asynchronous notifications are enabled for a server, the server will post the user-defined message to the window whenever there is a change in status or after a client has connected or disconnected from the server. The *wParam* message parameter will contain the notification message and the *lParam* message parameter will contain the handle to the server or the unique client ID. The following notification messages are defined:

Constant	Description
INET_NOTIFY_STARTUP	This notification is sent when the server has started and is preparing to accept client connections. This notification is only sent once, and only if asynchronous notifications are enabled immediately after the InetServerStart function is called. This message will not be sent once the server has begun accepting client connections or when notification messages are disabled and then subsequently re-enabled at a later time. The <i>lParam</i> message parameter will specify the handle to the server.
INET_NOTIFY_LISTEN	This notification is sent when the server is listening for

	<p>client connections. This notification message may be sent to the application multiple times over the lifetime of the server. If the server was suspended, this notification will be sent after the application calls the InetServerResume function to resume accepting client connections. The <i>lParam</i> message parameter will specify the handle to the server.</p>
INET_NOTIFY_SUSPEND	<p>This notification is sent when the server suspends accepting new connections because the application has called either the InetServerSuspend or InetServerSuspendEx function. This notification message may be sent to the application multiple times over the lifetime of the server. The <i>lParam</i> message parameter will specify the handle to the server.</p>
INET_NOTIFY_RESTART	<p>This notification is sent when the server is restarted using the InetServerRestart function. Note that the server socket handle provided by the <i>lParam</i> message parameter will specify the new socket handle of the restarted server instance, not the original socket handle. The <i>lParam</i> message parameter will specify the handle to the server.</p>
INET_NOTIFY_CONNECT	<p>This notification is sent when the server accepts a client connection and the thread that manages the client session has begun processing network events for that client. This message notification will not be sent if the client connection is rejected by the server. The <i>lParam</i> message parameter will specify the unique ID of the client that connected to the server.</p>
INET_NOTIFY_DISCONNECT	<p>This notification is sent when the client disconnects from the server and the client socket has been closed. This notification message may not occur for each client session that is forced to terminate as the result of the server being stopped using the InetServerStop function. The <i>lParam</i> message parameter will specify the unique ID of the client that disconnected from the server.</p>
INET_NOTIFY_SHUTDOWN	<p>This notification is sent when the server thread is in the process of terminating. At the time the application processes this notification message, the server handle in <i>lParam</i> will reference the defunct server and cannot be used with other server functions. The <i>lParam</i> message parameter will specify the handle to the server.</p>

If asynchronous notifications are enabled, you should never use those notifications as a replacement for an event handler. When an event occurs, the callback function that handles the event is invoked in the context of the thread that manages the client session. The application should exchange data with the client within that event handler and not in response to a notification message. These notification messages should only be used to update the application

UI in response to changes in the status of the server.

The INET_NOTIFY_CONNECT and INET_NOTIFY_DISCONNECT notifications are different from the other server notifications because the *lParam* message parameter does not specify the server handle, but rather the unique client ID associated with the session that connected to or disconnected from the server. If you need to obtain the handle to the client session using the ID, call the **InetGetClientHandle** function. To obtain the server handle in response to the INET_NOTIFY_CONNECT message, use the **InetGetClientServerById** function. Note that at the time the application processes the INET_NOTIFY_DISCONNECT notification message, the client session will have already terminated.

This function can only be used with a handle returned by the **InetServerStart** function and cannot be used with sockets created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[InetGetClientServerById](#), [InetGetServerStatus](#), [InetServerStart](#)

InetServerBroadcast Function

```
INT WINAPI InetServerBroadcast(  
    SOCKET hServer,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **InetServerBroadcast** function sends data to clients that are connected to the specified server.

Parameters

hServer

The socket handle.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server clients.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of clients that the data was sent to. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerBroadcast** function sends the contents of the buffer to all of the clients that are connected to the specified server. This function will block until all clients have been sent a copy of the data. There is no guarantee in which order the clients will receive and process the data that has been broadcast.

This function can only be used with a socket handle created using the **InetServerStart** function and cannot be used with sockets created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

See Also

[InetClientBroadcast](#), [InetWrite](#), [InetWriteLine](#)

InetServerLock Function

```
BOOL WINAPI InetServerLock(  
    SOCKET hServer  
);
```

The **InetServerLock** function locks the specified server, causing other client threads to block until it is unlocked.

Parameters

hServer

The socket handle to the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerLock** function causes the specified server to enter a locked state where only the current thread may interact with the server and the clients that are connected to it. While a server is locked, all other threads will block when they attempt to perform a network operation. When the server is unlocked, the blocked threads will resume normal execution.

This function should be used carefully, and a server should never be left in a locked state for an extended period of time. It is meant to be used when the server process updates a global data structure and it must prevent any other threads from performing a network operation during the update. Only one server can be locked at any one time, and once a server has been locked, it can only be unlocked by the same thread.

The program should always check the return value from this function, and should never assume that the lock has been established. If more than one thread attempts to lock a server at the same time, there is no guarantee as to which thread will actually establish the lock. If a potential deadlock situation is detected, this function will fail and return a value of zero.

Every time the **InetServerLock** function is called, an internal lock counter is incremented, and the lock will not be released until the lock count drops to zero. This means that each call to **InetServerLock** must be matched by an equal number of calls to the **InetServerUnlock** function. Failure to do so will result in the server becoming non-responsive as it remains in a locked state.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetGetLockedServer](#), [InetServerUnlock](#)

InetServerRestart Function

```
SOCKET WINAPI InetServerRestart(  
    SOCKET hServer  
);
```

The **InetServerRestart** function restarts the server, terminating all active client sessions.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is the new socket handle for the specified server. If the function fails, the return value is `INVALID_SOCKET`. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerRestart** function will restart the specified server, terminating all active client sessions and recreating the listening socket. The socket handle that is returned by the function is the handle for the new listening socket, and the old handle value is no longer valid. If the function is unable to recreate the listening socket for any reason, the server thread is terminated.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

See Also

[InetGetServerStatus](#), [InetServerResume](#), [InetServerStart](#), [InetServerStop](#), [InetServerSuspend](#)

InetServerResume Function

```
BOOL WINAPI InetServerResume(  
    SOCKET hServer  
);
```

The **InetServerResume** function resumes accepting client connections on the specified server.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerResume** function instructs the server to resume accepting client connections. Any pending client connections that were requested while the server was suspended will be accepted.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetGetServerStatus](#), [InetServerRestart](#), [InetServerStart](#), [InetServerStop](#), [InetServerSuspend](#)

InetServerStart Function

```
SOCKET WINAPI InetServerStart(  
    LPCTSTR lpszLocalHost,  
    UINT nLocalPort,  
    UINT nBacklog,  
    UINT nMaxClients,  
    UINT nTimeout,  
    UINT nPriority,  
    DWORD dwOptions,  
    INETEVENTPROC lpEventProc,  
    DWORD_PTR dwEventParam,  
    LPSECURITYCREDENTIALS lpCredentials  
);
```

The **InetServerStart** function begins listening for client connections on the specified local address and port number. The server is started in its own thread and manages the client sessions independently of the calling thread. All interaction with the server and its client sessions takes place inside the callback function specified by the caller.

Parameters

lpszLocalHost

A pointer to a string which specifies the local hostname or IP address address that the socket should be bound to. If this parameter is NULL or an empty string, then an appropriate address will automatically be used. A specific address should only be used if it is required by the application.

nLocalPort

The local port number that the socket should be bound to. This value must be greater than zero.

nBacklog

The maximum length of the queue allocated for pending client connections. A value of zero specifies that the size of the queue should be set to a maximum reasonable value. On Windows server platforms, the maximum value is large enough to queue several hundred pending connections.

nMaxClients

The maximum number of client connections that can be established with the server. A value of zero specifies that there should not be any fixed limit on the number of active client connections. This value can be adjusted after the server has been created by calling the **InetServerThrottle** function.

nTimeout

The number of seconds the server should wait for a client to perform a network operation. If the client does not exchange any information with the server within this period of time, a timeout event will occur. The timeout value affects all clients that are connected to the server.

nPriority

An integer value which specifies the priority for the server and all client sessions. The priority for a specific client session may be modified by calling the **InetSetClientPriority** function. This parameter may be one of the following values:

Constant	Description
----------	-------------

INET_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. It is recommended that most applications use this priority.
INET_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the client session. It is typically used with lightweight services running in the background that are designed for few client connections. The client thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
INET_PRIORITY_LOW	This priority lowers the overall resource utilization for the client session and meters the processor utilization for the client session. The client thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
INET_PRIORITY_HIGH	This priority increases the overall resource utilization for the client session and the thread will be given higher scheduling priority. It can be used when it is important for the client session thread to be highly responsive. It is not recommended that this priority be used on a system with a single processor.
INET_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The thread will be given higher scheduling priority and will be more responsive to the remote host. It is not recommended that this priority be used on a system with a single processor.

dwOptions

An unsigned integer used to specify one or more socket options. The following values are supported:

Constant	Description
INET_OPTION_NONE	No option specified. If the address and port number are in use by another application or a closed socket which was listening on this port is still in the TIME_WAIT state, the function will fail.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option enables a server application to listen for connections using the specified address and port number even if they were in use recently. This is typically used to enable an application to close the listening socket and immediately reopen it without getting an error that the address is in use.
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to

	keep the connection active. Enabling this option will also help applications detect the physical loss of a network connection, such as an Ethernet cable being unplugged. This option does not guarantee that persistent connections will be maintained over long periods of time.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and combines smaller packets into a single larger packet when sending data to a remote host. Specifying this option can improve the responsiveness and overall throughput of applications that implement their own buffering and exchange large amounts of information.
INET_OPTION_SECURE	This option specifies that a secure connection should be established with the client, where the client immediately initiates the SSL handshake when it connects to the server. To implement an explicit SSL session, where the client establishes a standard, non-secure connection and then sends a command to the server to initiate a secure session, you should not use this option. Instead, use the InetEnableSecurity function to selectively enable SSL for the client session.

lpEventProc

Specifies the address of the application defined callback function. For more information about the callback function, see the description of the **InetEventProc** callback function. This parameter cannot be NULL.

dwEventParam

A user-defined integer value that is passed to the callback function.

lpCredentials

Pointer to credentials structure [SECURITYCREDENTIALS](#). This may be NULL, unless the *dwOptions* parameter includes INET_OPTION_SECURE. When a secure session is specified, the fields *dwSize*, *lpszCertStore*, and *lpszCertName* must be defined, while other fields may be left undefined. Set *dwSize* to the size of the **SECURITYCREDENTIALS** structure.

Return Value

If the function succeeds, the return value is a socket handle. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

In most cases, the *lpszLocalHost* parameter should be a NULL pointer or an empty string. On a multihomed system, this will enable the server to accept connections on any appropriately configured network adapter. Specifying a hostname or IP address will limit client connections to that particular address. Note that the hostname or address must be one that is assigned to the local system, otherwise an error will occur.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the function will fail.

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address ":::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

If the `INET_OPTION_REUSEADDRESS` option is not specified, an error may be returned if a listening socket was recently created for the same local address and port number. By default, once a listening socket is closed there is a period of time that all applications must wait before the address can be reused (this is called the `TIME_WAIT` state). The actual amount of time depends on the operating system and configuration parameters, but is typically two to four minutes. Specifying this option enables an application to immediately re-use a local address and port number that was previously in use. Note that this does not permit more than one server to bind to the same address.

When the event handler callback function is invoked by the server, it normally executes in the context of the worker thread that manages that client session. This means that even if you do not explicitly create any threads in your application, you must design your program to be thread-safe, with synchronized access to global objects and data. If your application has a user interface, only the main UI thread should attempt to modify controls. If you attempt to modify a control from a worker thread, such as adding a row to a listbox control, it can result the application becoming deadlocked. This means that you should not attempt to directly update the UI from within the event handler function. To enable asynchronous server notifications for a GUI application, use the **`InetServerAsyncNotify`** function.

The socket handle returned by this function references the listening socket that was created when the server was started. The service is managed in another thread, and all interaction with the server and active client connections are performed inside the event handler. To disconnect all active connections, close the listening socket and terminate the server thread, call the **`InetServerStop`** function.

Example

```
#define SERVER_PORT      7000
#define SERVER_CLIENTS  100

SOCKET hServer = INVALID_SOCKET;

// Accept connections from clients that connection on port 7000 with a default
// backlog of 5 connections and a maximum of 100 client connections.

hServer = InetServerStart(NULL,
                          SERVER_PORT,
                          INET_BACKLOG,
                          SERVER_CLIENTS,
                          INET_TIMEOUT,
                          INET_PRIORITY_NORMAL,
                          INET_OPTION_REUSEADDRESS,
                          MyEventHandler,
                          0,
                          NULL);

if (hServer == INVALID_SOCKET)
{
    DWORD dwError;
```

```
TCHAR szError[256];

dwError = InetGetLastError();
InetGetErrorString(dwError, szError, 256);

MessageBox(NULL, szError, NULL, MB_OK|MB_TASKMODAL);
return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetGetServerData](#), [InetGetServerPriority](#), [InetGetServerStatus](#), [InetServerLock](#), [InetServerRestart](#), [InetServerResume](#), [InetServerStop](#), [InetServerSuspend](#), [InetServerThrottle](#), [InetServerUnlock](#), [InetSetServerData](#), [InetSetServerPriority](#)

InetServerStop Function

```
BOOL WINAPI InetServerStop(  
    SOCKET hServer  
);
```

The **InetServerStop** function signals the server to stop listening for connections and terminates all client sessions.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerStop** function signals the server to stop accepting client connections, disconnects all active client connections and terminates the thread that is managing the server session. The socket handle is no longer valid after the server has been stopped and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new server is started. An application should always consider the socket handle to be opaque and never depend on it being a specific value.

If this function is called when there is one or more clients connected to the server, it will signal each client thread to terminate and then wait for the server thread to terminate. As the client sessions are terminated, the event handler will not be invoked. If you wish to ensure that all clients are disconnected normally before stopping the server, call the **InetServerSuspendEx** function with the INET_SUSPEND_DISCONNECT option and then stop the server after the last client has disconnected.

Because the **InetServerStop** function waits for the server thread to terminate, this function may cause your application to block. If this is not desirable, use the **InetServerStopEx** function which can perform the shutdown sequence asynchronously.

After the server thread has been terminated, the listening socket will go into a TIME-WAIT state which prevents an application from reusing the same address and port number bound to that socket for a brief period of time, typically two to four minutes. This is normal behavior designed to prevent delayed or misrouted packets of data from being read by a subsequent connection. To immediately start a new server using the same local address and port number, the option INET_OPTION_REUSEADDRESS must be specified when calling the **InetServerStart** function.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

InetServerStopEx Function

```
BOOL WINAPI InetServerStopEx(  
    SOCKET hServer,  
    DWORD dwMilliseconds  
);
```

The **InetServerStopEx** function signals the server to stop listening for connections and terminates all client sessions.

Parameters

hServer

Handle to the server socket.

dwMilliseconds

An unsigned integer value that specifies the number of milliseconds to wait for all active clients to disconnect and the server thread to terminate.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerStopEx** function signals the server to stop accepting client connections, disconnects all active client connections and terminates the thread that is managing the server session. The socket handle is no longer valid after the server has been stopped and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new server is started. An application should always consider the socket handle to be opaque and never depend on it being a specific value.

Unlike the **InetServerStop** function, which waits for fixed period of time for the server thread to terminate, the **InetServerStopEx** allows the caller to determine how much time should be spent waiting for the clients to disconnect and the server thread to terminate. If the *dwMilliseconds* parameter has a value of INFINITE, the function will wait for an indefinite period of time until all clients have disconnected, the listening socket closed and the server thread has terminated. If the *dwMilliseconds* parameter has a value of zero, the function does not wait for the server to shutdown. Instead, it returns immediately and the shutdown process continues in the background.

If your application specifies a value of zero for the *dwMilliseconds* parameter, the event handler will be invoked with the INET_EVENT_DISCONNECT event as each client disconnects from the server during the shutdown process. If you depend on this event to perform some cleanup on a per-client basis, you must ensure that the application does not exit until the server thread has terminated. To perform a graceful shutdown of the server, it is recommended that you use the **InetServerSuspendEx** function and specify the INET_SUSPEND_DISCONNECT option. After all clients have disconnected, call the **InetServerStop** function to terminate the server thread.

After the server thread has been terminated, the listening socket will go into a TIME-WAIT state which prevents an application from reusing the same address and port number bound to that socket for a brief period of time, typically two to four minutes. This is normal behavior designed to prevent delayed or misrouted packets of data from being read by a subsequent connection. To immediately start a new server using the same local address and port number, the option INET_OPTION_REUSEADDRESS must be specified when calling the **InetServerStart** function.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cswsock10.h

Import Library: cs wskv10.lib

See Also

[InetGetServerStatus](#), [InetServerRestart](#), [InetServerStart](#), [InetServerStop](#), [InetServerSuspendEx](#)

InetServerSuspend Function

```
BOOL WINAPI InetServerSuspend(  
    SOCKET hServer  
);
```

Suspend accepting client connections on the specified server.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerSuspend** function instructs the server to suspend accepting new client connections. Any incoming client connections will be queued up to the maximum backlog value specified when the server was started. To resume accepting client connections, call the **InetServerResume** function.

It is recommended that you only suspend a server if absolutely necessary, and only for brief periods of time. If you want to limit the number of active client connections or control the connection rate for clients, use the **InetServerThrottle** function.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: csuskv10.lib

See Also

[InetGetServerStatus](#), [InetServerRestart](#), [InetServerResume](#), [InetServerStart](#), [InetServerStop](#), [InetServerThrottle](#)

InetServerSuspendEx Function

```
BOOL WINAPI InetServerSuspendEx(  
    SOCKET hServer,  
    DWORD dwOptions  
);
```

Suspend accepting client connections and optionally reject or disconnect clients.

Parameters

hServer

Handle to the server socket.

dwOptions

An unsigned integer that specifies one or more options.

Constant	Description
INET_SUSPEND_DEFAULT	Specifies that the server should suspend accepting new connections. New incoming client connections will be queued and clients that have already established a connection to the server will remain connected.
INET_SUSPEND_REJECT	Specifies that the server should suspend accepting new connections and reject any new client connections. Clients that have already established a connection to the server will remain connected.
INET_SUSPEND_DISCONNECT	Specifies that the server should suspend accepting new connections and disconnect all active clients that are currently connected to the server. If the INET_SUSPEND_REJECT option is also specified, new client connections will be rejected by the server.
INET_SUSPEND_WAIT	Specifies that the function should wait for the clients to disconnect from the server rather than return immediately to the caller. This option is only meaningful when used in conjunction with INET_SUSPEND_DISCONNECT.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerSuspendEx** function instructs the server to suspend accepting new client connections. Additional options can be used to control how the server suspends connections. If you plan on suspending the server for any period of time greater than a few seconds, it is recommended that you specify the INET_SUSPEND_REJECT option so that clients are not forced to wait.

If the INET_SUSPEND_DISCONNECT option is specified, the server will signal each client to disconnect and will stop accepting new connections. The event handler will be invoked for each client that disconnects. If the INET_SUSPEND_WAIT option is also specified, the function will wait

until the last client has disconnected from the server before returning to the caller. If there are a large number of clients connected to the server, this process may cause the application to block for an extended period of time and appear to be non-responsive to the user. For this reason, you should not specify the `INET_SUSPEND_WAIT` option if the function is being called from the application's main UI thread.

To perform a graceful shutdown of the server, it is recommended that you call **InetServerSuspendEx** with the `INET_SUSPEND_REJECT` and `INET_SUSPEND_DISCONNECT` options. This will allow each client to disconnect from the server and the server will reject any new incoming connections. After the last client has disconnected from the server, call the **InetServerStop** function to complete the shutdown process. The **InetGetClientThreads** function can be used to determine if all client session threads have terminated.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

See Also

[InetGetServerStatus](#), [InetServerRestart](#), [InetServerResume](#), [InetServerStart](#), [InetServerStop](#), [InetServerThrottle](#)

InetServerThrottle Function

```
BOOL WINAPI InetServerThrottle(  
    SOCKET hServer,  
    UINT nMaxClients,  
    UINT nMaxClientsPerAddress,  
    DWORD dwConnectionRate  
);
```

The **InetServerThrottle** function limits the number of active client connections, connections per address and connection rate.

Parameters

hServer

Handle to the server socket.

nMaxClients

A value which specifies the maximum number of clients that may connect to the server. A value of zero specifies that there is no fixed limit to the number of client connections.

nMaxClientsPerAddress

A value which specifies the maximum number of clients that may connect to the server from the same IP address. A value of zero specifies that there is no fixed limit to the number of client connections per address. By default, there is no limit on the number of client connections per address.

dwConnectionRate

A value which specifies a restriction on the rate of client connections, limiting the number of connections that will be accepted within that period of time. A value of zero specifies that there is no restriction on the rate of client connections. The higher this value, the fewer the number of connections that will be accepted within a specific period of time. By default, there is no limit on the client connection rate.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerThrottle** function is used to limit the number of connections and the connection rate to minimize the potential impact of a large number of client connections over a short period of time. This can be used to protect the server from a client application that is malfunctioning or a deliberate denial-of-service attack in which the attacker attempts to flood the server with connection attempts.

If the maximum number of client connections or maximum number of connections per address is exceeded, the server will reject subsequent connection attempts until the number of active client sessions drops below the specified threshold. Note that adjusting these values lower than the current connection limits will not affect clients that have already connected to the server. For example, if the **InetServerStart** function is called with the maximum number of clients set to 100, and then **InetServerThrottle** is called lowering that value to 75, no existing client connections will be affected by the change. However, the server will not accept any new connections until the number of active clients drops below 75.

Increasing the connection rate value will force the server to slow down the rate at which it will accept incoming client connection requests. For example, setting this parameter to a value of 1000 would limit the server to accepting one client connection every second, while a value of 250 would allow the server to accept four client connections per second. Note that significantly increasing the amount of time the server must wait to accept client connections can exceed the connection backlog queue, resulting in client connections being rejected.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

See Also

[InetGetServerStatus](#), [InetServerLock](#), [InetServerRestart](#), [InetServerResume](#), [InetServerStart](#), [InetServerSuspend](#), [InetServerUnlock](#)

InetServerUnlock Function

```
BOOL WINAPI InetServerUnlock(  
    SOCKET hServer  
);
```

The **InetServerUnlock** function unlock the specified server, allowing other client threads to resume execution.

Parameters

hServer

The socket handle to the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerUnlock** function releases the lock on the specified server and allows any blocked threads to resume execution. Only one server may be locked at any one time, and only the thread which established the lock can unlock the server.

Every time the **InetServerLock** function is called, an internal lock counter is incremented, and the lock will not be released until the lock count drops to zero. This means that each call to **InetServerLock** must be matched by an equal number of calls to the **InetServerUnlock** function. Failure to do so will result in the server becoming non-responsive as it remains in a locked state.

The program should always check the return value from this function, and should never assume that the lock has been released. If a potential deadlock situation is detected, this function will fail and return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

See Also

[InetGetLockedServer](#), [InetGetServerStatus](#), [InetServerLock](#)

InetSetClientData Function

```
BOOL WINAPI InetSetClientData(  
    SOCKET hClient,  
    VOID LpvData  
);
```

The **InetSetClientData** function sets the application defined data associated with the specified client session.

Parameters

hSocket

The socket handle.

lppvData

Pointer to the application defined data associated with the specified client session.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates that application defined data for the client session could not be modified. To get extended error information, call **InetGetLastError**.

Remarks

The **InetSetClientData** function is used to associate application defined data with a specific client session. This is typically used to associate a pointer to a data structure or a class instance with the client socket. A pointer to the data can be retrieved using the **InetGetClientData** function.

You should never specify a pointer to a local variable or data structure that will go out of scope when the calling function exits. If you do this, the pointer will no longer be valid after the function exits and attempting to dereference that pointer at some later time can cause an exception to be thrown and terminate the program. You should always allocate a block of memory for the data using a function such as **HeapAlloc** or **LocalAlloc**. If you specify the address of a static or global data structure, you must use thread synchronization functions when dereferencing and modifying that structure.

This function can only be used with client socket handles created using the SocketWrench server interface. It cannot be used with socket handles created using the **InetConnect** or **InetAccept** functions.

Example

```
UINT *pnValue1 = (UINT *)LocalAlloc(LPTR, sizeof(UINT));  
UINT *pnValue2 = NULL;  
  
*pnValue1 = 1234;  
  
if (InetSetClientData(hSocket, pnValue1) == FALSE)  
{  
    // Unable to associate the data with this session  
    return;  
}  
  
if (InetGetClientData(hSocket, &pnValue2) == FALSE)  
{  
    // Unable to retrieve the data associated with this session
```

```
    return;  
}  
  
// *pnValue2 == 1234  
printf("The value of user defined data is %u\n", *pnValue2);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetGetClientData](#), [InetGetServerData](#), [InetSetServerData](#)

InetSetClientMoniker Function

```
INT WINAPI InetSetClientMoniker(  
    SOCKET hSocket,  
    LPCTSTR LpszMoniker  
);
```

The **InetSetClientMoniker** function associates a unique string moniker with the specified client session.

Parameters

hSocket

Handle to the client socket.

lpszMoniker

Pointer to a string which specifies the moniker for the specified client socket. If this parameter is NULL or specifies an empty string, a moniker will no longer be associated with the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

A client moniker is a string which can be used to uniquely identify a specific client session aside from its socket handle. The **InetGetClientMoniker** function will return the moniker that was previously assigned to the client, if any. To obtain the socket handle associated with a given moniker, use the **InetFindClientMoniker** function.

Monikers are not case-sensitive, and they must be unique so that no client socket for a particular server can have the same moniker. The maximum length for a moniker is 127 characters.

The socket handle for the client must be one that was created as part of the SocketWrench server interface, and cannot be a socket that was created using the **InetConnect** or **InetAccept** functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetFindClientMoniker](#), [InetGetClientHandle](#), [InetGetClientId](#), [InetGetClientMoniker](#)

InetSetClientPriority Function

```
INT WINAPI InetSetClientPriority(  
    SOCKET hClient,  
    INT nPriority  
);
```

The **InetSetClientPriority** function sets the current priority for the specified client session.

Parameters

hClient

Handle to the client session.

nPriority

An integer value which specifies the new priority for the client session. It may be one of the following values:

Constant	Description
INET_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. It is recommended that most applications use this priority.
INET_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the client session. It is typically used with lightweight services running in the background that are designed for few client connections. The client thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
INET_PRIORITY_LOW	This priority lowers the overall resource utilization for the client session and meters the processor utilization for the client session. The client thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
INET_PRIORITY_HIGH	This priority increases the overall resource utilization for the client session and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
INET_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The client thread will be given higher scheduling priority and will be more responsive to network events. It is not recommended that this priority be used on a system with a single processor.

Return Value

If the function succeeds, the return value is the previous priority for the specified client session. If the function fails, the return value is INET_ERROR. To get extended error information, call

InetGetLastError.

Remarks

The **InetSetClientPriority** function can be used to change the current priority assigned to the specified client session. The client priority is inherited from the priority specified when the server is started using the **InetServerStart** function.

The socket handle for the client must be one that was created as part of the SocketWrench server interface, and cannot be a socket that was created using the **InetConnect** or **InetAccept** functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetGetClientPriority](#), [InetGetServerPriority](#), [InetServerStart](#), [InetSetServerPriority](#)

InetSetHostFile Function

```
INT WINAPI InetSetHostFile(  
    LPCTSTR lpszFileName  
);
```

The **InetSetHostFile** function specifies the name of an alternate file to use when resolving hostnames and IP addresses. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **InetGetHostAddress** and **InetGetHostNames** function. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

```
ipaddress hostname [hostalias ...]
```

For example, one typical entry maps the name "localhost" to the local loopback IP address. This would be entered as:

```
127.0.0.1 localhost
```

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

Parameters

lpszFileName

Pointer to a string that specifies the name of the file. If the parameter is NULL, then the current host file is cleared from the cache and only the default host file will be used to resolve hostnames and addresses.

Return Value

If the function succeeds, the return value is the number of entries in the host file. A return value of INET_ERROR indicates failure. To get extended error information, call **InetGetLastError**.

Remarks

This function loads the file into memory allocated for the current thread. If the contents of the file have changed after the function has been called, those changes will not be reflected when resolving hostnames or addresses. To reload the host file from disk, call this function again with the same file name. To remove the alternate host file from memory, specify a NULL pointer as the parameter.

If a host file has been specified, it is processed before the default host file when resolving a hostname into an IP address, or an IP address into a hostname. If the host name or address is not found, or no host file has been specified, a nameserver lookup is performed.

To determine if an alternate host file has been specified, use the **InetGetHostFile** function. A return value of zero indicates that no alternate host file has been cached for the current thread.

A system may have a default host file, which is used to resolve hostnames before performing a nameserver lookup. To determine the name of this file, use the **InetGetDefaultHostFile** function. It is not necessary to specify this default host file, since it is always used to resolve host names and addresses.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetGetDefaultHostFile](#), [InetGetHostAddress](#), [InetGetHostFile](#), [InetGetHostName](#)

InetSetLastError Function

```
VOID WINAPI InetSetLastError(  
    DWORD dwErrorCode  
);
```

The **InetSetLastError** function sets the last-error code for the caller. This function is typically used to clear the last error by passing a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last-error code for the caller.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SOCKET or INET_ERROR. Those functions which call **InetSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the [InetGetLastError](#) function. The use of **InetGetLastError** is optional; an application can call it to find out the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[InetGetErrorString](#), [InetGetLastError](#)

InetSetOption Function

```
INT WINAPI InetSetOption(  
    SOCKET hSocket,  
    DWORD dwOption,  
    BOOL bEnabled  
);
```

The **InetSetOption** function is used to enable or disable a specific socket option.

Parameters

hSocket

The socket handle.

dwOption

An unsigned integer used to specify one of the socket options. These options cannot be combined. The following values are recognized:

Constant	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_KEEPAIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option is commonly used by server applications.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.

bEnabled

A boolean flag. If the flag is set to a non-zero value, the option is enabled. Otherwise the socket option is disabled.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

It is not recommend that you disable the Nagle algorithm by specifying the `INET_OPTION_NODELAY` flag unless it is absolutely required. Doing so can have a significant, negative impact on the performance of the application and network.

If if the `INET_OPTION_KEEPAIVE` option is enabled, keep-alive packets will start being generated five seconds after the socket has become idle with no data being sent or received. Enabling this option can be used by applications to detect when a physical network connection has been lost. However, it is recommended that most applications query the remote host directly to determine if the connection is still active. This is typically accomplished by sending specific commands to the

server to query its status, or checking the elapsed time since the last response from the server.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

See Also

[InetAsyncConnectEx](#), [InetConnectEx](#), [InetGetOption](#)

InetSetServerData Function

```
BOOL WINAPI InetSetServerData(  
    SOCKET hServer,  
    VOID LpvData  
);
```

The **InetSetServerData** function sets the application defined data associated with the specified server.

Parameters

hSocket

The socket handle.

lppvData

Pointer to the application defined data associated with the specified server.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates that application defined data for the server could not be modified. To get extended error information, call **InetGetLastError**.

Remarks

The **InetSetServerData** function is used to associate application defined data with a specific server. A pointer to the data can be retrieved using the **InetGetServerData** function.

You should never specify a pointer to a local variable or data structure that will go out of scope when the calling function exits. If you do this, the pointer will no longer be valid after the function exits and attempting to dereference that pointer at some later time can cause an exception to be thrown and terminate the program. You should always allocate a block of memory for the data using a function such as **HeapAlloc** or **LocalAlloc**. If you specify the address of a static or global data structure, you must use thread synchronization functions when dereferencing and modifying that structure.

This function can only be used with server socket handles created using the **InetServerStart** function. It cannot be used with socket handles created using the **InetListen** or **InetListenEx** functions.

Example

```
UINT *pnValue1 = (UINT *)LocalAlloc(LPTR, sizeof(UINT));  
UINT *pnValue2 = NULL;  
  
*pnValue1 = 1234;  
  
if (InetSetServerData(hServer, pnValue1) == FALSE)  
{  
    // Unable to associate the data with this server  
    return;  
}  
  
if (InetGetServerData(hServer, &pnValue2) == FALSE)  
{  
    // Unable to retrieve the data associated with this server  
    return;  
}
```

```
// *pnValue2 == 1234  
printf("The value of user defined data is %u\n", *pnValue2);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: csuskv10.lib

See Also

[InetGetClientData](#), [InetGetServerData](#), [InetSetClientData](#)

InetSetServerPriority Function

```
INT WINAPI InetSetServerPriority(  
    SOCKET hServer,  
    INT nPriority  
);
```

The **InetSetServerPriority** function sets the current priority for the specified server.

Parameters

hServer

Handle to the server socket.

nPriority

An integer value which specifies the new priority for the server. It may be one of the following values:

Constant	Description
INET_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
INET_PRIORITY_LOW	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
INET_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
INET_PRIORITY_HIGH	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
INET_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Return Value

If the function succeeds, the return value is the previous priority assigned to the server. If the function fails, the return value is INET_PRIORITY_INVALID. To get extended error information, call **InetGetLastError**.

Remarks

The **InetSetServerPriority** function can be used to change the current priority assigned to the specified server. Client connections that are accepted after this function is called will inherit the new priority as their default priority. Previously existing client connections will not be affected by this function. To modify the priority for an active client session, use the **InetSetClientPriority** function.

Higher priority values increase the thread priority and processor utilization for each client session. You should only change the server priority if you understand the impact it will have on the system and have thoroughly tested your application. Configuring the server to run with a higher priority can have a negative effect on the performance of other programs running on the system.

The socket handle for the server must be one that was created using the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: csuskv10.lib

See Also

[InetGetClientPriority](#), [InetGetServerPriority](#), [InetServerStart](#), [InetSetClientPriority](#)

InetSetServerStackSize Function

```
BOOL WINAPI InetSetServerStackSize(  
    SOCKET hServer,  
    DWORD dwStackSize  
);
```

Change the initial size of the stack allocated for threads created by the server.

Parameters

hServer

Handle to the server socket.

dwStackSize

The amount of memory that will be committed to the stack for each thread created by the server. If this value is zero, a default stack size will be used.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetSetServerStackSize** function changes the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes. Increasing or decreasing the stack size will only affect new threads that are created by the server, it will not affect those threads that have already been created to manage active client sessions. It is recommended that most applications use the default stack size.

You should not change the stack size unless you understand the impact that it will have on your system and have thoroughly tested your application. Increasing the initial commit size of the stack will remove pages from the total system commit limit, and every page of memory that is reserved for stack cannot be used for any other purpose.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetGetServerStackSize](#), [InetServerStart](#)

InetSetTimeout Function

```
INT WINAPI InetSetTimeout(  
    SOCKET hSocket,  
    UINT nTimeout  
);
```

The **InetSetTimeout** function sets the interval that is used when waiting for a blocking operation to complete.

Parameters

hSocket

Handle to the socket.

nTimeout

Duration of timeout interval, in seconds. If a value over 1000 is specified, it is assumed that milliseconds are intended by the user, and the value actually used will be adjusted accordingly.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetGetTimeout](#), [InetConnect](#), [InetAccept](#), [InetIsReadable](#), [InetIsWritable](#)

InetShutdown Function

```
INT WINAPI InetShutdown(  
    SOCKET hSocket,  
    DWORD dwOption  
);
```

The **InetShutdown** function is used to disable reception or transmission of data, or both.

Parameters

hSocket

The socket handle.

dwOption

An unsigned integer used to specify one of the shutdown options. These options cannot be combined. The following values are recognized:

Value	Constant	Description
0	INET_SHUTDOWN_READ	Disable reception of data.
1	INET_SHUTDOWN_WRITE	Disable transmission of data.
2	INET_SHUTDOWN_BOTH	Disable both reception and transmission of data.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

This function is rarely needed. It is provided as an interface to the Windows Sockets **shutdown** function.

In some asynchronous applications, it may be desirable for a client to inform the server that no further communication is wanted, while allowing the client to read any residual data that may reside in internal buffers on the client side. **InetShutdown** accomplishes this because the socket handle is still valid after it has been called, although some or all communication with the remote host has ceased.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetDisconnect](#)

InetStoreStream Function

```
BOOL WINAPI InetStoreStream(  
    SOCKET hSocket,  
    LPCTSTR lpszFileName,  
    DWORD dwLength,  
    LPDWORD lpdwCopied  
    DWORD dwOffset,  
    DWORD dwOptions  
);
```

The **InetStoreStream** function reads the socket data stream and stores the contents in the specified file.

Parameters

hSocket

The socket handle. The socket must reference a stream socket, not a datagram or raw socket. If the socket type is not valid, the function will return an error.

lpszFileName

Pointer to a string which specifies the name of the file to create or overwrite.

dwLength

An unsigned integer which specifies the maximum number of bytes to read from the socket and write to the file. If this value is zero, then the function will continue to read data from the socket until the remote host disconnects or an error occurs.

lpdwCopied

A pointer to an unsigned integer value which will contain the number of bytes written to the file when the function returns.

dwOffset

An unsigned integer which specifies the byte offset into the file where the function will start storing data read from the socket. Note that all data after this offset will be truncated. A value of zero specifies that the file should be completely overwritten if it already exists.

dwOptions

An unsigned integer value which specifies one or more options. Programs can use a bitwise operator to combine any of the following values:

Constant	Description
INET_STREAM_CONVERT	The data stream is considered to be textual and will be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data written to the file to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a

	carriage-return character before each linefeed.
INET_STREAM_UNICODE	The data stream should be converted to Unicode. This option should only be used with text data, and will result in the stream data being written as 16-bit wide characters rather than 8-bit bytes. The amount of data returned will be twice the amount read from the source data stream. If the <i>dwOffset</i> parameter has a value of zero, the Unicode byte order mark (BOM) will be written to the beginning of the file.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetStoreStream** function enables an application to read an arbitrarily large stream of data and store it in a file. This function is essentially a simplified version of the **InetReadStream** function, designed specifically to be used with files rather than memory buffers or handles.

This function will force the thread to block until the operation completes. If this function is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to asynchronous operation when it has finished. If another socket operation is attempted while **InetStoreStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this function only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

Because **InetStoreStream** can potentially cause the application to block for long periods of time as the data stream is being read, the function will periodically generate INET_EVENT_PROGRESS events. An application can register an event handler using the **InetRegisterEvent** function, and can obtain information about the current operation by calling the **InetGetStreamInfo** function.

Example

```
DWORD dwCopied;
BOOL bResult;

bResult = InetStoreStream(hSocket,
                          lpzFileName,
                          &dwCopied,
                          0,
                          INET_STREAM_CONVERT);

if (bResult && dwCopied > 0)
{
    // The data has been written to the file
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

Import Library: `cswskv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetGetStreamInfo](#), [InetRead](#), [InetReadLine](#), [InetReadStream](#), [InetWrite](#), [InetWriteLine](#),
[InetWriteStream](#)

InetUninitialize Function

```
VOID WINAPI InetUninitialize();
```

The **InetUninitialize** function terminates the use of the library.

Parameters

None.

Return Value

None.

Remarks

An application is required to perform a successful **InetInitialize** call before it can call any of the other library functions. When it has completed the use of library, the application must call **InetUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There must be a call to **InetUninitialize** for every successful call to **InetInitialize** made by a process. In a multithreaded environment, operations for all threads are terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cswsock10.h`

Import Library: `cswskv10.lib`

See Also

[InetDisconnect](#), [InetInitialize](#)

InetWrite Function

```
INT WINAPI InetWrite(  
    SOCKET hSocket,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **InetWrite** function sends the specified number of bytes to the remote host.

Parameters

hSocket

The socket handle.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the remote host.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the application to send the remaining data at some later point. For non-blocking connections, the program must wait for the INET_EVENT_WRITE asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cswskv10.lib

See Also

[InetFlush](#), [InetRead](#), [InetReadEx](#), [InetWriteEx](#)

InetWriteEx Function

```
INT WINAPI InetWriteEx(  
    SOCKET hSocket,  
    LPVOID lpvBuffer,  
    INT cbBuffer,  
    DWORD dwReserved,  
    LPINTERNET_ADDRESS lpRemoteAddress,  
    UINT nRemotePort  
);
```

The **InetWriteEx** function sends the specified number of bytes to the remote host.

Parameters

hSocket

The socket handle.

lpvBuffer

The pointer to the buffer which contains the data that is to be sent to the remote host.

cbBuffer

The number of bytes to send from the specified buffer.

dwReserved

Reserved parameter. This value must always be zero.

lpRemoteAddress

Pointer to an [INTERNET_ADDRESS](#) structure that specifies the address of the remote host that is to receive the data being written. For TCP stream sockets, this parameter must always be NULL or specify the same address that was used to establish the connection. For UDP datagram sockets, this may specify any valid IP address.

nRemotePort

The port number of the remote host that is to receive the data being written. For TCP stream sockets, this value must always be zero, or specify the same port number that was used to establish the connection. For UDP datagram sockets, this may specify any valid port number.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the application to send the remaining data at some later point. For non-blocking connections, the program must wait for the `INET_EVENT_WRITE` asynchronous notification message before it resumes sending data.

This function extends the **InetWrite** function to additional information about the destination IP address and port number for the data being written. For a client TCP connection, the IP address and remote port must be the same values that were used to establish the connection. When writing on a UDP socket, this is the IP address and remote port of the peer that will receive the datagram. This information can be used in conjunction with the **InetReadEx** function to send a datagram back to that host.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetFlush](#), [InetRead](#), [InetReadEx](#), [InetWrite](#), [INTERNET_ADDRESS](#)

InetWriteLine Function

```
BOOL WINAPI InetWriteLine(  
    SOCKET hSocket,  
    LPCTSTR lpszBuffer,  
    LPINT lpnLength  
);
```

The **InetWriteLine** function sends a line of text to the remote host, terminated by a carriage-return and linefeed.

Parameters

hSocket

The socket handle. The socket must reference a stream socket, not a datagram or raw socket. If the socket type is not valid, the function will return an error.

lpszBuffer

The pointer to a string buffer which contains the data that will be sent to the remote host. All characters up to, but not including, the terminating null character will be written to the socket. The data will always be terminated with a carriage-return and linefeed control character sequence. If this parameter points to an empty string or NULL pointer, then a only a carriage-return and linefeed are written to the socket.

lpnLength

A pointer to an integer value which will contain the number of characters written to the socket, including the carriage-return and linefeed sequence. If this information is not required, a NULL pointer may be specified.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetWriteLine** function writes a line of text to the remote host and terminates the line with a carriage-return and linefeed control character sequence. Unlike the **InetWrite** function which writes arbitrary bytes of data to the socket, this function is specifically designed to write a single line of text data from a string.

If the *lpszBuffer* string is terminated with a linefeed (LF) or carriage return (CR) character, it will be automatically converted to a standard CRLF end-of-line sequence. Because the string will be sent with a terminating CRLF sequence, the value returned in the *lpnLength* parameter will typically be larger than the original string length (reflecting the additional CR and LF characters), unless the string was already terminated with CRLF.

There are some limitations when using **InetWriteLine**. The function should only be used to send text, never binary data. In particular, the function will discard nulls and append linefeed and carriage return control characters to the data stream. The Unicode version of this function will accept a Unicode string, however this function does not support writing raw Unicode data to the socket. Unicode strings will be automatically converted to UTF-8 encoding using the **WideCharToMultiByte** function and then written to the socket as a stream of bytes.

This function will force the thread to block until the complete line of text has been written, the write operation times out or the remote host aborts the connection. If this function is called with

asynchronous events enabled, it will automatically switch the socket into a blocking mode, send the data and then restore the socket to asynchronous operation. If another socket operation is attempted while **InetWriteLine** is blocked sending data to the remote host, an error will occur. It is recommended that this function only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **InetWrite** and **InetWriteLine** function calls can be safely intermixed.

Unlike the **InetWrite** function, it is possible for data to have been written to the socket if the return value is zero. For example, if a timeout occurs while the function is waiting to send more data to the remote host, it will return zero; however, some data may have already been written prior to the error condition. If this is the case, the *lpnLength* argument will specify the number of characters actually written up to that point.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[InetIsWritable](#), [InetRead](#), [InetReadLine](#), [InetWrite](#)

InetWriteStream Function

```
BOOL WINAPI InetWriteStream(  
    SOCKET hSocket,  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwOptions  
);
```

The **InetWriteStream** function writes data from the stream buffer to the specified socket.

Parameters

hSocket

The socket handle. The socket must reference a stream socket, not a datagram or raw socket. If the socket type is not valid, the function will return an error.

lpvBuffer

Pointer to the buffer that contains or references the data to be written to the socket. The actual argument depends on the value of the *dwOptions* parameter which specifies how the data stream will be accessed.

lpdwLength

A pointer to an unsigned integer value which specifies the size of the buffer and contains the number of bytes written when the function returns. This argument should always point to an initialized value. If the *lpvBuffer* argument specifies a memory buffer or global memory handle, then this argument cannot point to an initialized value of zero.

dwOptions

An unsigned integer value which specifies the stream buffer type to be used when writing the data stream to the socket. One of the following stream types may be specified:

Constant	Description
INET_STREAM_DEFAULT	The default stream buffer type is determined by the value passed as the <i>lpvBuffer</i> parameter. If the argument specifies a global memory handle, then the function will write the data referenced by that handle; otherwise, the function will consider the parameter a pointer to a block of memory which contains data to be written. In most cases, it is recommended that an application explicitly specify the stream buffer type rather than using the default value.
INET_STREAM_MEMORY	The <i>lpvBuffer</i> argument specifies a pointer to a block of memory which contains the data to be written to the socket. If this stream buffer type is used, the <i>lpdwLength</i> argument must point to an unsigned integer which has been initialized with the size of the buffer.
INET_STREAM_HGLOBAL	The <i>lpvBuffer</i> argument specifies a global memory handle that references the data to be written to the socket. The handle must have been created by a

	call to the GlobalAlloc or GlobalReAlloc function. If this stream buffer type is used, the <i>lpdwLength</i> argument must point to an unsigned integer which has been initialized with the size of the buffer.
INET_STREAM_HANDLE	The <i>lpvBuffer</i> argument specifies a Windows handle to an open file, console or pipe. This should be the same handle value returned by the CreateFile function in the Windows API. The data read using the ReadFile function with this handle will be written to the socket.
INET_STREAM_SOCKET	The <i>lpvBuffer</i> argument specifies a socket handle. The data read from the socket specified by this handle will be written to the socket specified by the <i>hSocket</i> parameter. The socket handle passed to this function must have been created by this library; if it is a socket created by a third-party library or directly by the Windows Sockets API, you should either attach the socket using the InetAttachSocket function or use the INET_STREAM_HANDLE stream buffer type instead.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetWriteStream** function enables an application to write an arbitrarily large stream of data from memory or a file to the specified socket. Unlike the **InetWrite** function, which may not write all of the data in a single function call, **InetWriteStream** will only return when all of the data has been written or an error occurs.

This function will force the thread to block until the operation completes. If this function is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, write the data stream and then restore the socket to asynchronous operation when it has finished. If another socket operation is attempted while **InetWriteStream** is blocked sending data to the remote host, an error will occur. It is recommended that this function only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

It is possible for some data to have been written even if the function returns a value of zero. Applications should also check the value of the *lpdwLength* argument to determine if any data was sent. For example, if a timeout occurs while the function is waiting to write more data, it will return zero; however, some data may have already been written to the socket prior to the error condition.

Because **InetWriteStream** can potentially cause the application to block for long periods of time as the data stream is being written, the function will periodically generate INET_EVENT_PROGRESS events. An application can register an event handler using the **InetRegisterEvent** function, and can obtain information about the current operation by calling the **InetGetStreamInfo** function.

Example

```
HANDLE hFile;
DWORD dwLength;

hFile = CreateFile(lpszFileName,
                  GENERIC_READ,
                  FILE_SHARE_READ,
                  NULL,
                  OPEN_EXISTING,
                  FILE_FLAG_SEQUENTIAL_SCAN,
                  NULL);

if (hFile == INVALID_HANDLE_VALUE)
    return;

dwLength = GetFileSize(hFile, NULL);

if (dwLength > 0)
{
    BOOL bResult = InetWriteStream(
        hSocket,
        hFile,
        &dwLength,
        INET_STREAM_HANDLE);
}

CloseHandle(hFile);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Import Library: cssock10.lib

See Also

[InetGetStreamInfo](#), [InetRead](#), [InetReadLine](#), [InetReadStream](#), [InetStoreStream](#), [InetWrite](#), [InetWriteLine](#)

SocketWrench Data Structures

- INETSTREAMINFO
- INITDATA
- INTERNET_ADDRESS
- SECURITYCREDENTIALS
- SECURITYINFO

INETSTREAMINFO Structure

This structure contains information about the data stream being currently read or written.

```
typedef struct _INETSTREAMINFO
{
    DWORD    dwStreamThread;
    DWORD    dwStreamSize;
    DWORD    dwStreamCopied;
    DWORD    dwStreamMode;
    DWORD    dwStreamError;
    DWORD    dwBytesPerSecond;
    DWORD    dwTimeElapsed;
    DWORD    dwTimeEstimated;
} INETSTREAMINFO, *LPINETSTREAMINFO;
```

Members

dwStreamThread

Specifies the numeric ID for the thread that created the socket.

dwStreamSize

The maximum number of bytes that will be read or written. This is the same value as the buffer length specified by the caller, and may be zero which indicates that no maximum size was specified. Note that if this value is zero, the application will be unable to calculate a completion percentage or estimate the amount of time for the operation to complete.

dwStreamCopied

The total number of bytes that have been copied to or from the stream buffer.

dwStreamMode

A numeric value which specifies the stream operation that is current being performed. It may be one of the following values:

Constant	Description
INET_STREAM_READ	Data is being read from the socket and stored in the specified stream buffer.
INET_STREAM_WRITE	Data is being written from the specified stream buffer to the socket.

dwStreamError

The last error that occurred when reading or writing the data stream. If no error has occurred, this value will be zero.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the operation is completed. This is based on the average number of bytes transferred per second and requires that a maximum stream buffer size be specified by the caller.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cssock10.h`

See Also

[InetReadStream](#), [InetStoreStream](#), [InetWriteStream](#)

INITDATA Structure

This structure contains information about the SocketTools library when the initialization function returns.

```
typedef struct _INITDATA
{
    DWORD      dwSize;
    DWORD      dwVersionMajor;
    DWORD      dwVersionMinor;
    DWORD      dwVersionBuild;
    DWORD      dwOptions;
    DWORD_PTR  dwReserved1;
    DWORD_PTR  dwReserved2;
    TCHAR      szDescription[128];
} INITDATA, *LPINITDATA;
```

Members

dwSize

Size of this structure. This structure member must be set to the size of the structure prior to calling the initialization function. Failure to do so will cause the function to fail.

dwVersionMajor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the major version number for the library.

dwVersionMinor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the minor version number for the library.

dwVersionBuild

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the build number for the library.

dwOptions

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved1

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved2

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

szDescription

A null terminated string which describes the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Unicode: Implemented as Unicode and ANSI versions.

INTERNET_ADDRESS Structure

This structure represents a numeric IPv4 or IPv6 address in network byte order.

```
typedef struct _INTERNET_ADDRESS
{
    INT    ipFamily;
    BYTE   ipNumber[16];
} INTERNET_ADDRESS, *LPINTERNET_ADDRESS;
```

Members

ipFamily

An integer which identifies the type of IP address. It will be one of the following values:

Constant	Description
INET_ADDRESS_UNKNOWN	The address has not been specified or the bytes in the <i>ipNumber</i> array does not represent a valid address. Functions which populate this structure will use this value to indicate that the address cannot be determined.
INET_ADDRESS_IPV4	Specifies that the address is in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address is in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

ipNumber

A byte array which contains the numeric form of the IP address. This array is large enough to store both IPv4 (32 bit) and IPv6 (128 bit) addresses. The values are stored in network byte order.

Remarks

The **INTERNET_ADDRESS** structure is used by some functions to represent an Internet address in a binary format that is compatible with both IPv4 and IPv6 addresses. Applications that use this structure should consider it to be opaque, and should not modify the contents of the structure directly.

For compatibility with legacy applications that expect an IP address to be 32 bits and stored in an unsigned integer, you can copy the first four bytes of the *ipNumber* array using the **CopyMemory** function or equivalent. Note that if this is done, your application should always check the *ipFamily* member first to make sure that it is actually an IPv4 address.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cswsock10.h

SECURITYCREDENTIALS Structure

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on

	what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that

will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A pointer to a null terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A pointer to a null terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A pointer to a null terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A pointer to a null terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
------------	-------------

CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
----	---

MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
----	--

ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.
------	--

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

A pointer to a null terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used for SSH connections

and should always be NULL when establishing a secure connection using SSL or TLS.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* member should be set to CREDENTIAL_STORE_FILENAME and the *lpzCertStore* member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS#12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the *lpzPassword* member or the library will be unable to access the certificate.

Note that the *lpzUserName* and *lpzPassword* members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

The TLS 1.1 and TLS 1.2 protocols are only supported on Windows 7, Windows Server 2008 R2 and later versions of the platform. If these options are specified and the application is running on Windows XP or Windows Vista, the protocol version will be downgraded to TLS 1.0 for backwards compatibility with those platforms.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cssock10.h

Unicode: Implemented as Unicode and ANSI versions.

SECURITYINFO Structure

This structure contains information about a secure connection that has been established.

```
typedef struct _SECURITYINFO
{
    DWORD        dwSize;
    DWORD        dwProtocol;
    DWORD        dwCipher;
    DWORD        dwCipherStrength;
    DWORD        dwHash;
    DWORD        dwHashStrength;
    DWORD        dwKeyExchange;
    DWORD        dwCertStatus;
    SYSTEMTIME   stCertIssued;
    SYSTEMTIME   stCertExpires;
    LPCTSTR      lpszCertIssuer;
    LPCTSTR      lpszCertSubject;
    LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to **sizeof(SECURITYINFO)** prior to passing the address of this structure to the function. Note that if this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the protocol that was selected to establish the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The

	<p>correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.</p>
SECURITY_PROTOCOL_TLS10	<p>The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS11	<p>The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS12	<p>The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.</p>

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_CIPHER_RC2	The RC2 block cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
SECURITY_CIPHER_DES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher, using 56-bit

	keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher (also known as the Rijndael cipher) is a fixed block size cipher which use a key size of 128, 192 or 256 bits. This cipher is supported on Windows XP SP3 and later versions of the operating system.
SECURITY_CIPHER_SKIPJACK	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
SECURITY_CIPHER_BLOWFISH	The Blowfish block cipher was selected. This is a variable key length cipher up to 448 bits, using a 64-bit block size.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected. One of the following values will be returned:

Constant	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected. One of the following values will be returned:

--	--

Constant	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography. This key exchange algorithm is only supported on Windows XP SP3 and later versions of the operating system.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server. This member only has meaning for connections using the SSL or TLS protocols. One of the following values will be returned:

Constant	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertIssuer

A pointer to a string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertSubject

A pointer to a string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszFingerprint

A pointer to a string which contains a sequence of hexadecimal values that uniquely identify the server. This member is only used when a connection has been established using the Secure Shell (SSH) protocol.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Unicode: Implemented as Unicode and ANSI versions.

SocketWrench Library Error Codes

Value	Constant	Description
0x80042711	ST_ERROR_NOT_HANDLE_OWNER	Handle not owned by the current thread
0x80042712	ST_ERROR_FILE_NOT_FOUND	The specified file or directory does not exist
0x80042713	ST_ERROR_FILE_NOT_CREATED	The specified file could not be created
0x80042714	ST_ERROR_OPERATION_CANCELED	The blocking operation has been canceled
0x80042715	ST_ERROR_INVALID_FILE_TYPE	The specified file is a block or character device, not a regular file
0x80042716	ST_ERROR_INVALID_DEVICE	The specified device or address does not exist
0x80042717	ST_ERROR_TOO_MANY_PARAMETERS	The maximum number of function parameters has been exceeded
0x80042718	ST_ERROR_INVALID_FILE_NAME	The specified file name contains invalid characters or is too long
0x80042719	ST_ERROR_INVALID_FILE_HANDLE	Invalid file handle passed to function
0x8004271A	ST_ERROR_FILE_READ_FAILED	Unable to read data from the specified file
0x8004271B	ST_ERROR_FILE_WRITE_FAILED	Unable to write data to the specified file
0x8004271C	ST_ERROR_OUT_OF_MEMORY	Out of memory
0x8004271D	ST_ERROR_ACCESS_DENIED	Access denied
0x8004271E	ST_ERROR_INVALID_PARAMETER	Invalid argument passed to function
0x8004271F	ST_ERROR_CLIPBOARD_UNAVAILABLE	The system clipboard is currently unavailable
0x80042720	ST_ERROR_CLIPBOARD_EMPTY	The system clipboard is empty or does not contain any text data
0x80042721	ST_ERROR_FILE_EMPTY	The specified file does not contain any data
0x80042722	ST_ERROR_FILE_EXISTS	The specified file already exists
0x80042723	ST_ERROR_END_OF_FILE	End of file
0x80042724	ST_ERROR_DEVICE_NOT_FOUND	The specified device could not be found
0x80042725	ST_ERROR_DIRECTORY_NOT_FOUND	The specified directory could not be found
0x80042726	ST_ERROR_INVALID_BUFFER	Invalid memory address passed to function
0x80042728	ST_ERROR_NO_HANDLES	No more handles available to this process
0x80042733	ST_ERROR_OPERATION_WOULD_BLOCK	The specified operation would block the current thread
0x80042734	ST_ERROR_OPERATION_IN_PROGRESS	A blocking operation is currently in progress
0x80042735	ST_ERROR_ALREADY_IN_PROGRESS	The specified operation is already in progress
0x80042736	ST_ERROR_INVALID_HANDLE	Invalid handle passed to function

0x80042737	ST_ERROR_INVALID_ADDRESS	Invalid network address specified
0x80042738	ST_ERROR_INVALID_SIZE	Datagram is too large to fit in specified buffer
0x80042739	ST_ERROR_INVALID_PROTOCOL	Invalid network protocol specified
0x8004273A	ST_ERROR_PROTOCOL_NOT_AVAILABLE	The specified network protocol is not available
0x8004273B	ST_ERROR_PROTOCOL_NOT_SUPPORTED	The specified protocol is not supported
0x8004273C	ST_ERROR_SOCKET_NOT_SUPPORTED	The specified socket type is not supported
0x8004273D	ST_ERROR_INVALID_OPTION	The specified option is invalid
0x8004273E	ST_ERROR_PROTOCOL_FAMILY	Specified protocol family is not supported
0x8004273F	ST_ERROR_PROTOCOL_ADDRESS	The specified address is invalid for this protocol family
0x80042740	ST_ERROR_ADDRESS_IN_USE	The specified address is in use by another process
0x80042741	ST_ERROR_ADDRESS_UNAVAILABLE	The specified address cannot be assigned
0x80042742	ST_ERROR_NETWORK_UNAVAILABLE	The networking subsystem is unavailable
0x80042743	ST_ERROR_NETWORK_UNREACHABLE	The specified network is unreachable
0x80042744	ST_ERROR_NETWORK_RESET	Network dropped connection on remote reset
0x80042745	ST_ERROR_CONNECTION_ABORTED	Connection was aborted due to timeout or other failure
0x80042746	ST_ERROR_CONNECTION_RESET	Connection was reset by remote network
0x80042747	ST_ERROR_OUT_OF_BUFFERS	No buffer space is available
0x80042748	ST_ERROR_ALREADY_CONNECTED	Connection already established with remote host
0x80042749	ST_ERROR_NOT_CONNECTED	No connection established with remote host
0x8004274A	ST_ERROR_CONNECTION_SHUTDOWN	Unable to send or receive data after connection shutdown
0x8004274C	ST_ERROR_OPERATION_TIMEOUT	The specified operation has timed out
0x8004274D	ST_ERROR_CONNECTION_REFUSED	The connection has been refused by the remote host
0x80042750	ST_ERROR_HOST_UNAVAILABLE	The specified host is unavailable
0x80042751	ST_ERROR_HOST_UNREACHABLE	Remote host is unreachable
0x80042753	ST_ERROR_TOO_MANY_PROCESSES	Too many processes are using the networking subsystem
0x8004276B	ST_ERROR_NETWORK_NOT_READY	Network subsystem is not ready for communication
0x8004276C	ST_ERROR_INVALID_VERSION	This version of the operating system is not supported

0x8004276D	ST_ERROR_NETWORK_NOT_INITIALIZED	The networking subsystem has not been initialized
0x80042775	ST_ERROR_REMOTE_SHUTDOWN	The remote host has initiated a graceful shutdown sequence
0x80042AF9	ST_ERROR_INVALID_HOSTNAME	The specified hostname is invalid or could not be resolved
0x80042AFA	ST_ERROR_HOSTNAME_NOT_FOUND	The specified hostname could not be found
0x80042AFB	ST_ERROR_HOSTNAME_REFUSED	Unable to resolve hostname, request refused
0x80042AFC	ST_ERROR_HOSTNAME_NOT_RESOLVED	Unable to resolve hostname, no address for specified host
0x80042EE1	ST_ERROR_INVALID_LICENSE	The license for this product is invalid
0x80042EE2	ST_ERROR_PRODUCT_NOT_LICENSED	This product is not licensed to perform this operation
0x80042EE3	ST_ERROR_NOT_IMPLEMENTED	This function has not been implemented on this platform
0x80042EE4	ST_ERROR_UNKNOWN_LOCALHOST	Unable to determine local host name
0x80042EE5	ST_ERROR_INVALID_HOSTADDRESS	Invalid host address specified
0x80042EE6	ST_ERROR_INVALID_SERVICE_PORT	Invalid service port number specified
0x80042EE7	ST_ERROR_INVALID_SERVICE_NAME	Invalid or unknown service name specified
0x80042EE8	ST_ERROR_INVALID_EVENTID	Invalid event identifier specified
0x80042EE9	ST_ERROR_OPERATION_NOT_BLOCKING	No blocking operation in progress on this socket
0x80042F45	ST_ERROR_SECURITY_NOT_INITIALIZED	Unable to initialize security interface for this process
0x80042F46	ST_ERROR_SECURITY_CONTEXT	Unable to establish security context for this session
0x80042F47	ST_ERROR_SECURITY_CREDENTIALS	Unable to open client certificate store or establish client credentials
0x80042F48	ST_ERROR_SECURITY_CERTIFICATE	Unable to validate the certificate chain for this session
0x80042F49	ST_ERROR_SECURITY_DECRYPTION	Unable to decrypt data stream
0x80042F4A	ST_ERROR_SECURITY_ENCRYPTION	Unable to encrypt data stream
0x80043031	ST_ERROR_MAXIMUM_CONNECTIONS	The maximum number of client connections exceeded
0x80043032	ST_ERROR_THREAD_CREATION_FAILED	Unable to create a new thread for the current process
0x80043033	ST_ERROR_INVALID_THREAD_HANDLE	The specified thread handle is no longer valid
0x80043034	ST_ERROR_THREAD_TERMINATED	The specified thread has been terminated
0x80043035	ST_ERROR_THREAD_DEADLOCK	The operation would result in the current

		thread becoming deadlocked
0x80043036	ST_ERROR_INVALID_CLIENT_MONIKER	The specified moniker is not associated with any client session
0x80043037	ST_ERROR_CLIENT_MONIKER_EXISTS	The specified moniker has been assigned to another client session
0x80043038	ST_ERROR_SERVER_INACTIVE	The specified server is not listening for client connections
0x80043039	ST_ERROR_SERVER_SUSPENDED	The specified server is suspended and not accepting client connections

Telnet Protocol Library

Establish an interactive terminal session with a server.

Reference

- [Functions](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

File Name	CSTNTV10.DLL
Version	10.0.1468.2518
LibID	A081335C-B907-4179-949F-FCE154F7C456
Import Library	CSTNTV10.LIB
Dependencies	None
Standards	RFC 854

Overview

The Telnet protocol is used to establish a connection with a server which provides a virtual terminal session for a user. Its functionality is similar to how character based consoles and serial terminals work, enabling a user to login to the server, execute commands and interact with applications running on the server. The class provides an interface for establishing the connection, negotiating certain options (such as whether characters will be echoed back to the client) and handling the standard I/O functions needed by the program.

The API also includes functions that enable a program to easily scan the data stream for specific sequences of characters, making it very simple to write light-weight client interfaces to applications running on the server. This library can be combined with the SocketTools Terminal Emulation API to provide complete terminal emulation services for a standard ANSI or DEC-VT220 terminal.

This library supports secure connections using the standard SSL and TLS protocols. To establish a secure connection to the server using the Secure Shell (SSH) protocol, use the SocketTools Secure Shell API.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Telnet Protocol Functions

Function	Description
TelnetAbort	Abort the current session and close the connection with the server
TelnetAsyncConnect	Connect asynchronously to the specified server
TelnetAttachThread	Attach the specified client handle to another thread
TelnetBreak	Send a break signal to the server
TelnetCancel	Cancel the current blocking operation
TelnetConnect	Connect to the specified server
TelnetCreateSecurityCredentials	Create a new security credentials structure
TelnetDeleteSecurityCredentials	Delete a previously created security credentials structure
TelnetDisableEvents	Disable asynchronous event notification
TelnetDisableTrace	Disable logging of socket function calls to the trace log
TelnetDisconnect	Disconnect from the current server
TelnetEnableEvents	Enable asynchronous event notification
TelnetEnableTrace	Enable logging of socket function calls to a file
TelnetEventProc	Callback function that processes events generated by the client
TelnetFreezeEvents	Suspend asynchronous event processing
TelnetGetErrorString	Return a description for the specified error code
TelnetGetLastError	Return the last error code
TelnetGetMode	Return the current client mode
TelnetGetSecurityInformation	Return security information about the current client connection
TelnetGetStatus	Return the current client status
TelnetGetTerminalType	Return the current terminal type
TelnetGetTimeout	Return the number of seconds until an operation times out
TelnetInitialize	Initialize the library and validate the specified license key at runtime
TelnetIsBlocking	Determine if the client is blocked, waiting for information
TelnetIsConnected	Determine if the client is connected to the server
TelnetIsReadable	Determine if data can be read from the server
TelnetIsThere	Determine if the server is available
TelnetIsWritable	Determine if data can be written to the server
TelnetLogin	Login to the server using the specified username and password
TelnetRead	Read data returned by the server
TelnetReadLine	Read a line of text from the server and return it in a string buffer
TelnetRegisterEvent	Register an event callback function

TelnetSearch	Search for a specific character sequence in the data stream
TelnetSetLastError	Set the last error code
TelnetSetMode	Set the current client mode
TelnetSetTerminalType	Set the current terminal type
TelnetSetTimeout	Set the number of seconds until an operation times out
TelnetUninitialize	Terminate use of the library by the application
TelnetWrite	Write data to the server
TelnetWriteLine	Write a line of text to the server

TelnetAbort Function

```
INT WINAPI TelnetAbort(  
    HCLIENT hClient  
);
```

The **TelnetAbort** function aborts the current session and terminates the connection.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Remarks

When the **TelnetAbort** function is called, the Telnet abort sequence is sent to the server and the connection to the server is terminated. Once this function returns, the client handle is no longer valid. If a program is currently executing on the server at the time this function is called, that program may be terminated as a result of the session being aborted. Applications should normally call **TelnetDisconnect** to gracefully disconnect from the server and should only use this function when the connection must be aborted immediately.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstntv10.lib

See Also

[TelnetBreak](#), [TelnetCancel](#), [TelnetIsBlocking](#)

TelnetAsyncConnect Function

```
HCLIENT WINAPI TelnetAsyncConnect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSECURITYCREDENTIALS LpCredentials,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **TelnetAsyncConnect** function is used to establish a connection with the server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **TelnetConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TELNET_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
TELNET_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.

TELNET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
TELNET_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using either the SSL or TLS protocol.
TELNET_OPTION_SECURE_EXPLICIT	This option specifies the client should attempt to establish a secure connection with the server using the START_TLS option. The client initiates a standard connection with the server, then requests a secure connection during the option negotiation process.
TELNET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
TELNET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
TELNET_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpCredentials

Pointer to credentials structure [SECURITYCREDENTIALS](#). This parameter is only used if the TELNET_OPTION_SECURE option is specified for the connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields *dwSize*, *lpCertStore*, and *lpCertName* must be defined, while other fields may be left undefined. Set *dwSize* to the size of the **SECURITYCREDENTIALS** structure.

hEventWnd

The handle to the event notification window. This window receives messages which notify the

client of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **TelnetGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
TELNET_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
TELNET_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
TELNET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
TELNET_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
TELNET_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
TELNET_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To cancel asynchronous notification and return the client to a blocking mode, use the **TelnetDisableEvents** function.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that

created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **TelnetAttachThread** function.

Specifying the TELNET_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TelnetConnect](#), [TelnetCreateSecurityCredentials](#), [TelnetDeleteSecurityCredentials](#),
[TelnetDisconnect](#), [TelnetInitialize](#), [TelnetUninitialize](#)

TelnetAttachThread Function

```
DWORD WINAPI TelnetAttachThread(  
    HCLIENT hClient  
    DWORD dwThreadId  
);
```

The **TelnetAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is `TELNET_ERROR`. To get extended error information, call

TelnetGetLastError.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **TelnetAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **TelnetAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **TelnetCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **TelnetUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstntv10.lib`

See Also

[TelnetCancel](#), [TelnetAsyncConnect](#), [TelnetConnect](#), [TelnetDisconnect](#), [TelnetUninitialize](#)

TelnetBreak Function

```
INT WINAPI TelnetBreak(  
    HCLIENT hClient  
);
```

The **TelnetBreak** function sends a signal to the server which may terminate an application that is currently running. The actual response to the break signal depends on the application.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TelnetAbort](#), [TelnetCancel](#), [TelnetRead](#), [TelnetWrite](#)

TelnetCancel Function

```
INT WINAPI TelnetCancel(  
    HCLIENT hClient  
);
```

The **TelnetCancel** function cancels any outstanding blocking operation in the client, causing the blocking function to fail. The application may then retry the operation or terminate the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Remarks

When the **TelnetCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[TelnetAbort](#), [TelnetBreak](#), [TelnetIsBlocking](#)

TelnetConnect Function

```
HCLIENT WINAPI TelnetConnect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSECURITYCREDENTIALS LpCredentials  
);
```

The **TelnetConnect** function is used to establish a connection with the server.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TELNET_OPTION_NONE	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
TELNET_OPTION_TUNNEL	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
TELNET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the SSL or TLS protocols.
TELNET_OPTION_SECURE	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure

	connections using either the SSL or TLS protocol.
TELNET_OPTION_SECURE_EXPLICIT	This option specifies the client should attempt to establish a secure connection with the server using the START_TLS option. The client initiates a standard connection with the server, then requests a secure connection during the option negotiation process.
TELNET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
TELNET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
TELNET_OPTION_FREETHREAD	This option specifies the handle returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the handle is synchronized across multiple threads.

lpCredentials

Pointer to credentials structure [SECURITYCREDENTIALS](#). This parameter is only used if the TELNET_OPTION_SECURE option is specified for the connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields *dwSize*, *lpCertStore*, and *lpCertName* must be defined, while other fields may be left undefined. Set *dwSize* to the size of the **SECURITYCREDENTIALS** structure.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **TelnetGetLastError**.

Remarks

To prevent this function from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **TelnetConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

The *dwOptions* argument can be used to specify the threading model that is used by the library

when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the it is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **TelnetAttachThread** function.

Specifying the TELNET_OPTION_FREETHREAD option enables any thread to call any function using the handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the handle is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same handle.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TelnetCreateSecurityCredentials](#), [TelnetDeleteSecurityCredentials](#), [TelnetDisconnect](#), [TelnetInitialize](#), [TelnetUninitialize](#)

TelnetCreateSecurityCredentials Function

```
BOOL WINAPI TelnetCreateSecurityCredentials(  
    DWORD dwProtocol,  
    DWORD dwOptions,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName,  
    LPVOID lpvReserved,  
    LPSECURITYCREDENTIALS* lppCredentials  
);
```

The **TelnetCreateSecurityCredentials** function creates a **SECURITYCREDENTIALS** structure.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is

	supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A pointer to a string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A pointer to a string which specifies the certificate owner's password. A value of NULL specifies

that no password is required. This parameter is only used if a PKCS12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A pointer to a string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

lpvReserved

Pointer reserved for future use. Set it to NULL when using this function.

lppCredentials

Pointer to an [LPSECURITYCREDENTIALS](#) pointer. The memory for the credentials structure will be allocated by this function and must be released by calling the **TelnetDeleteSecurityCredentials** function when it is no longer needed. The pointer value must be set to NULL before the function is called. It is important to note that this is a pointer to a pointer variable, not a pointer to the SECURITYCREDENTIALS structure itself.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **TelnetGetLastError**.

Remarks

The structure that is created by this function may be used as client credentials when establishing a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can

be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **TelnetAsyncConnect** or **TelnetConnect** functions.

Example

```
LPSECURITYCREDENTIALS lpSecCred = NULL;
TelnetCreateSecurityCredentials(SEcurity_PROTOCOL_DEFAULT,
                               0,
                               NULL,
                               NULL,
                               lpzCertStore,
                               lpzCertName,
                               NULL,
                               &lpSecCred);

hClient = TelnetConnect(lpzHostName,
                       TELNET_PORT_SECURE,
                       TELNET_TIMEOUT,
                       TELNET_OPTION_SECURE,
                       lpSecCred);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TelnetAsyncConnect](#), [TelnetConnect](#), [TelnetDeleteSecurityCredentials](#),
[TelnetGetSecurityInformation](#), [SECURITYCREDENTIALS](#)

TelnetDeleteSecurityCredentials Function

```
VOID WINAPI TelnetDeleteSecurityCredentials(  
    LPSECURITYCREDENTIALS* lppCredentials  
);
```

The **TelnetDeleteSecurityCredentials** function deletes an existing **SECURITYCREDENTIALS** structure.

Parameters

lppCredentials

Pointer to an **LPSECURITYCREDENTIALS** pointer. On exit from the function, the pointer will be NULL.

Return Value

None.

Example

```
if (lpSecCred)  
    TelnetDeleteSecurityCredentials(&lpSecCred);  
  
TelnetUninitialize();
```

Remarks

This function can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TelnetCreateSecurityCredentials](#), [TelnetUninitialize](#)

TelnetDisableEvents Function

```
INT WINAPI TelnetDisableEvents(  
    HCLIENT hClient  
);
```

The **TelnetDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `TELNET_ERROR`. To get extended error information, call **TelnetGetLastError**.

Remarks

This function affects both event notification and event callbacks. Any outstanding events in the message queue should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstntv10.lib`

See Also

[TelnetEnableEvents](#), [TelnetFreezeEvents](#), [TelnetRegisterEvent](#)

TelnetDisableTrace Function

```
BOOL WINAPI TelnetDisableTrace();
```

The **TelnetDisableTrace** function disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstntv10.lib

See Also

[TelnetEnableTrace](#)

TelnetDisconnect Function

```
INT WINAPI TelnetDisconnect(  
    HCLIENT hClient  
);
```

The **TelnetDisconnect** function terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TelnetAsyncConnect](#), [TelnetConnect](#), [TelnetUninitialize](#)

TelnetEnableEvents Function

```
INT WINAPI TelnetEnableEvents(  
    HCLIENT hClient,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **TelnetEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **TelnetRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the event notification window. This window receives a user-defined message which specifies the event that has occurred. If this value is NULL, event notification is disabled.

uEventMsg

An unsigned integer which specifies the user-defined message that is sent when an event occurs. This parameter's value must be greater than the value of WM_USER. If the *hEventWnd* parameter is NULL, this value must be specified as WM_NULL.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
TELNET_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
TELNET_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
TELNET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.

TELNET_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
TELNET_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
TELNET_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

As noted, some events are only generated when the client is asynchronous mode. These events depend on the Windows Sockets asynchronous notification mechanism.

If event notification is disabled by specifying a NULL window handle, there may still be outstanding events in the message queue that must be processed. Since event handling has been disabled, these events should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[TelnetDisableEvents](#), [TelnetFreezeEvents](#), [TelnetRegisterEvent](#)

TelnetEnableTrace Function

```
BOOL WINAPI TelnetEnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **TelnetEnableTrace** function enables the logging of Windows Sockets function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace function logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TelnetDisableTrace](#)

TelnetEventProc Function

```
VOID CALLBACK TelnetEventProc(  
    HCLIENT hClient,  
    UINT nEvent,  
    DWORD dwError,  
    DWORD_PTR dwParam  
);
```

The **TelnetEventProc** function is an application-defined callback function that processes events generated by the client.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **TelnetRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **TelnetRegisterEvent** function. The **TelnetEventProc** function is a placeholder for the application-defined function name.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TelnetDisableEvents](#), [TelnetEnableEvents](#), [TelnetFreezeEvents](#), [TelnetRegisterEvent](#)

TelnetFreezeEvents Function

```
INT WINAPI TelnetFreezeEvents(  
    HCLIENT hClient,  
    BOOL bFreeze  
);
```

The **TelnetFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[TelnetDisableEvents](#), [TelnetEnableEvents](#), [TelnetRegisterEvent](#)

TelnetGetErrorString Function

```
INT WINAPI TelnetGetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);
```

The **TelnetGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **TelnetGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstntv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TelnetGetLastError](#), [TelnetSetLastError](#)

TelnetGetLastError Function

```
DWORD WINAPI TelnetGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **TelnetSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **TelnetGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **TelnetSetLastError(0)** when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or TELNET_ERROR. Those functions which call **TelnetSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[TelnetGetErrorString](#), [TelnetSetLastError](#)

TelnetGetMode Function

```
UINT WINAPI TelnetGetMode(  
    HCLIENT hClient  
);
```

The **TelnetGetMode** function returns the current client mode.

Parameters

hClient

A handle to the client session.

Return Values

If the function succeeds, the return value is the current client mode. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Remarks

The client mode is a combination of one or more flags which determines how the client handles local character echo and character processing. The following values are recognized:

Value	Description
TELNET_MODE_LOCALECHO	The local client is responsible for echoing data entered by the user. By default, this mode is not set which means that the server is responsible for echoing back each character written to it.
TELNET_MODE_BINARY	Data exchanged between the client and server should not be converted or line buffered. If this option is not specified, the high-bit will be cleared on all characters and single linefeeds will be automatically converted to carriage-return/linefeed sequences.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TelnetSetMode](#)

TelnetGetSecurityInformation Function

```
BOOL WINAPI TelnetGetSecurityInformation(  
    HCLIENT hClient,  
    LPSECURITYINFO lpSecurityInfo  
);
```

The **TelnetGetSecurityInformation** function returns security protocol, encryption and certificate information about the current client connection.

Parameters

hClient

Handle to the client session.

lpSecurityInfo

A pointer to a [SECURITYINFO](#) structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **TelnetGetLastError**.

Remarks

This function is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value `SECURITY_PROTOCOL_NONE`.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;  
  
securityInfo.dwSize = sizeof(SECURITYINFO);  
if (TelnetGetSecurityInformation(hClient, &securityInfo))  
{  
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)  
    {  
        MessageBox(NULL, _T("The connection is not secure"),  
                    _T("Connection"), MB_OK);  
    }  
    else  
    {  
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)  
        {  
            MessageBox(NULL, _T("The connection is secure"),  
                        _T("Connection"), MB_OK);  
        }  
        else  
        {  
            MessageBox(NULL, _T("The server certificate not valid"),  
                        _T("Connection"), MB_OK);  
        }  
    }  
}
```

```
}  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TelnetConnect](#), [TelnetDisconnect](#), [SECURITYINFO](#)

TelnetGetStatus Function

```
INT WINAPI TelnetGetStatus(  
    HCLIENT hClient  
);
```

The **TelnetGetStatus** function returns the current status of the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Remarks

The **TelnetGetStatus** function returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Constant	Description
1	TELNET_STATUS_IDLE	The client is current idle and not sending or receiving data.
2	TELNET_STATUS_CONNECT	The client is establishing a connection with the server.
3	TELNET_STATUS_READ	The client is reading data from the server.
4	TELNET_STATUS_WRITE	The client is writing data to the server.
5	TELNET_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[TelnetIsBlocking](#), [TelnetIsConnected](#), [TelnetIsReadable](#), [TelnetIsWritable](#)

TelnetGetTerminalType Function

```
INT WINAPI TelnetGetTerminalType(  
    HCLIENT hClient,  
    LPCTSTR lpszTermType,  
    INT nMaxLength  
);
```

The **TelnetGetTerminalType** function returns the terminal type for the current client session.

Parameters

hClient

Handle to the client session.

lpszTermType

Points to a buffer which the current terminal type is copied into. This buffer should be at least 32 characters in length, including the terminating null character.

nMaxLength

Maximum number of characters that may be copied to the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is the length of the terminal type name. A value of zero indicates that no terminal type has been specified. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TelnetSetTerminalType](#)

TelnetGetTimeout Function

```
INT WINAPI TelnetGetTimeout(  
    HCLIENT hClient  
);
```

The **TelnetGetTimeout** function returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[TelnetSetTimeout](#)

TelnetInitialize Function

```
BOOL WINAPI TelnetInitialize(  
    LPCTSTR lpszLicenseKey,  
    LPINITDATA lpData  
);
```

The **TelnetInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpData

Pointer to an INITDATA data structure. This parameter may be NULL if the initialization data for the library is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **TelnetGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must insure that this function is called with a valid license key.

If the *lpData* argument is specified, it must point to an INITDATA structure which has been initialized by setting the *dwSize* member to the size of the structure. All other structure members should be set to zero. If the function is successful, then the INITDATA structure will be filled with identifying information about the library.

Although it is only required that **TelnetInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **TelnetUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **TelnetInitialize** or **TelnetUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TelnetAsyncConnect](#), [TelnetConnect](#), [TelnetDisconnect](#), [TelnetUninitialize](#)

TelnetIsBlocking Function

```
BOOL WINAPI TelnetIsBlocking(  
    HCLIENT hClient  
);
```

The **TelnetIsBlocking** function is used to determine if the client is currently performing a blocking operation.

Parameters

hClient

Handle to the client session.

Return Value

If the client is performing a blocking operation, the function returns a non-zero value. If the client is not performing a blocking operation, or the client handle is invalid, the function returns zero.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking function may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **TelnetIsBlocking** function can be used to determine if the client is already blocked, and if so, take some other action such as warning the user that they must wait for the operation to complete.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[TelnetCancel](#)

TelnetIsConnected Function

```
BOOL WINAPI TelnetIsConnected(  
    HCLIENT hClient  
);
```

The **TelnetIsConnected** function is used to determine if the client is currently connected to a server.

Parameters

hClient

Handle to the client session.

Return Value

If the client is connected to a server, the function returns a non-zero value. If the client is not connected, or the client handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TelnetIsBlocking](#), [TelnetIsReadable](#), [TelnetIsThere](#), [TelnetIsWritable](#)

TelnetIsReadable Function

```
BOOL WINAPI TelnetIsReadable(  
    HCLIENT hClient,  
    INT nTimeout,  
    LPDWORD lpdwAvail  
);
```

The **TelnetIsReadable** function is used to determine if data is available to be read from the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the specified timeout period, the function returns a non-zero value. If the client cannot read any data, the function returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[TelnetGetStatus](#), [TelnetIsBlocking](#), [TelnetIsConnected](#), [TelnetIsThere](#), [TelnetIsWritable](#), [TelnetRead](#)

TelnetIsThere Function

```
BOOL WINAPI TelnetIsThere(  
    HCLIENT hClient  
);
```

The **TelnetIsThere** function reports the response of the client to a "Are you there" command to the telnet server.

Parameters

hClient

Handle to the client session.

Return Value

The function returns a non-zero value if the server acknowledges a specific control sequence used to determine if a Telnet server is responsive. If the server does not respond, the function will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[TelnetGetStatus](#), [TelnetIsBlocking](#), [TelnetIsConnected](#), [TelnetIsReadable](#), [TelnetIsWritable](#)

TelnetIsWritable Function

```
BOOL WINAPI TelnetIsWritable(  
    HCLIENT hClient,  
    INT nTimeout  
);
```

The **TelnetIsWritable** function is used to determine if data can be written to the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If the client can write data to the server within the specified timeout period, the function returns a non-zero value. If the client cannot write any data, the function returns zero.

Remarks

Although this function can be used to determine if some amount of data can be sent to the remote process it does not indicate the amount of data that can be written without blocking the client. In most cases, it is recommended that large amounts of data be broken into smaller logical blocks, typically some multiple of 512 bytes in length.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstntv10.lib`

See Also

[TelnetGetStatus](#), [TelnetIsBlocking](#), [TelnetIsConnected](#), [TelnetIsReadable](#), [TelnetIsThere](#), [TelnetWrite](#)

TelnetLogin Function

```
BOOL WINAPI TelnetLogin(  
    HCLIENT hClient,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword  
);
```

The **TelnetLogin** function attempts to authenticate the user and log them in to the current session.

Parameters

hClient

Handle to the client session.

lpszUserName

A pointer to a string which specifies the name of the user to authenticate.

lpszPassword

A pointer to a string which specifies the password to be used when authenticating the user. If the user does not require a password, this parameter may be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **TelnetGetLastError**.

Remarks

The **TelnetLogin** function is used to authenticate a user, logging them into the server. This function is specifically designed to work with most UNIX based servers, and may work with other servers that use a similar login process. The function works by scanning the data stream for a username prompt and then replying with the specified username. If that is successful, it will then scan for a password prompt and provide the specified password. If no recognized prompt is found, or if the server responds with an error indicating that the username or password is invalid, the function will fail.

If the **TelnetLogin** function succeeds, the next call to **TelnetRead** by the client will return any welcome message to the user. This is typically followed by a command prompt where the user can enter commands to be executed on the server. The data sent by the server during the login process is discarded and not available when the function returns. If the client requires this information, use the **TelnetSearch** function to automate the login process instead.

Because the **TelnetLogin** function is designed for UNIX based systems, it may not work with servers running on other operating system platforms such as Windows or VMS. In this case, applications should use the **TelnetSearch** function to search for the appropriate login prompts in the data stream.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TelnetAsyncConnect](#), [TelnetConnect](#), [TelnetIsConnected](#), [TelnetIsReadable](#), [TelnetRead](#),
[TelnetSearch](#)

TelnetRead Function

```
INT WINAPI TelnetRead(  
    HCLIENT hClient,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **TelnetRead** function reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hClient

Handle to the client session.

lpBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Remarks

When **TelnetRead** is called and the client is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TelnetIsReadable](#), [TelnetSearch](#), [TelnetWrite](#)

TelnetReadLine Function

```
BOOL WINAPI TelnetReadLine(  
    HCLIENT hClient,  
    LPTSTR lpszBuffer,  
    LPINT lpnLength  
);
```

The **TelnetReadLine** function reads up to a line of data and returns it in a string buffer.

Parameters

hSocket

Handle to the client session.

lpszBuffer

Pointer to the string buffer that will contain the data when the function returns. The string will be terminated with a null byte, and will not contain the end-of-line characters.

lpnLength

A pointer to an integer value which specifies the length of the buffer. The value should be initialized to the maximum number of characters that can be copied into the string buffer, including the terminating null character. When the function returns, its value will be updated with the actual length of the string.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **TelnetGetLastError**.

Remarks

The **TelnetReadLine** function reads data sent by the server and copies it into a specified string buffer. Unlike the **TelnetRead** function which reads arbitrary bytes of data, this function is specifically designed to return a single line of text data in a string. When an end-of-line character sequence is encountered, the function will stop and return the data up to that point. The string buffer is guaranteed to be null-terminated and will not contain the end-of-line characters.

There are some limitations when using **TelnetReadLine**. The function should only be used to read text, never binary data. In particular, the function will discard nulls, linefeed and carriage return control characters. The Unicode version of this function will return a Unicode string, however this function does not support reading raw Unicode data from the server. The data is internally buffered as octets (eight-bit bytes) and converted to Unicode using the **MultiByteToWideChar** function.

This function will force the thread to block until an end-of-line character sequence is processed, the read operation times out or the server closes its end of the connection. If this function is called with asynchronous events enabled, it will automatically switch the client into a blocking mode, read the data and then restore the client to asynchronous operation. If another client operation is attempted while **TelnetReadLine** is blocked waiting for data from the server, an error will occur. It is recommended that this function only be used with blocking (synchronous) client connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **TelnetRead** and **TelnetReadLine** function calls can be intermixed, however be aware that **TelnetRead** will consume any data that has already been buffered by the **TelnetReadLine**

function and this may have unexpected results.

Unlike the **TelnetRead** function, it is possible for data to be returned in the buffer even if the return value is zero. Applications should also check the value of the *lpnLength* argument to determine if any data was copied into the buffer. For example, if a timeout occurs while the function is waiting for more data to arrive, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the function return value.

Example

```
TCHAR szBuffer[MAXBUFLEN];
INT nLength;
BOOL bResult;

do
{
    nLength = sizeof(szBuffer);
    bResult = TelnetReadLine(hSocket, szBuffer, &nLength);

    if (nLength > 0)
    {
        // Process the line of data returned in the string
        // buffer; the string is always null-terminated
    }
} while (bResult);

DWORD dwError = TelnetGetLastError();
if (dwError == ST_ERROR_CONNECTION_CLOSED)
{
    // The server has closed its side of the connection and
    // there is no more data available to be read
}
else if (dwError != 0)
{
    // An error has occurred while reading a line of data
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TelnetIsReadable](#), [TelnetRead](#), [TelnetWrite](#), [TelnetWriteLine](#)

TelnetRegisterEvent Function

```
INT WINAPI TelnetRegisterEvent(  
    HCLIENT hClient,  
    UINT nEvent,  
    TELNETEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

The **TelnetRegisterEvent** function registers a callback function for the specified event.

Parameters

hClient

Handle to the client session.

nEvent

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Constant	Description
TELNET_EVENT_CONNECT	The connection to the server has completed.
TELNET_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
TELNET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
TELNET_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
TELNET_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
TELNET_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **TelnetEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `TELNET_ERROR`. To get extended error information, call **TelnetGetLastError**.

Remarks

The **TelnetRegisterEvent** function associates a callback function with a specific event. The event handler is an **TelnetEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the *lpEventProc* parameter must be declared using the `__stdcall` calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstntv10.lib`

See Also

[TelnetDisableEvents](#), [TelnetEnableEvents](#), [TelnetEventProc](#), [TelnetFreezeEvents](#)

TelnetSearch Function

```
BOOL WINAPI TelnetSearch(  
    HCLIENT hClient,  
    LPCTSTR lpszString,  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwReserved  
);
```

The **TelnetSearch** function searches for a specific character sequence in the data stream and stops reading if the sequence is encountered.

Parameters

hClient

Handle to the client session.

lpszString

A pointer to a string which specifies the sequence of characters to search for in the data stream. This parameter cannot be NULL or point to an empty string.

lpvBuffer

A pointer to a byte buffer which will contain the output from the server, or a pointer to a global memory handle which will reference the output when the function returns. If the output from the server is not required, this parameter may be NULL.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvBuffer* parameter. If the *lpvBuffer* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual number of bytes of output stored in the buffer. If the *lpvBuffer* parameter is NULL, this parameter should also be NULL.

dwReserved

A reserved parameter. This value must be zero.

Return Value

If the function succeeds and the character sequences was found in the data stream, the return value is non-zero. If the function fails or a timeout occurs before the sequence is found, the return value is zero. To get extended error information, call **TelnetGetLastError**.

Remarks

The **TelnetSearch** function searches for a character sequence in the data stream and stops reading when it is found. This is useful when the client wants to automate responses to the server, such as logging in a user and executing a command. The function collects the output from the server and stores it in the buffer specified by the *lpvBuffer* parameter. When the function returns, the buffer will contain everything sent by the server up to and including the search string.

The *lpvBuffer* parameter may be specified in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the a fixed amount of output. In this case, the *lpvBuffer* parameter will point to the buffer that was allocated, the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer. If the server sends more output than can be stored in the buffer, the remaining output will be discarded.


```

        0);
    }

    // If the shell prompt was found, issue the command
    // and capture the output into the hgblBuffer global
    // memory buffer; the cbBuffer variable will contain
    // the actual number of bytes in the buffer when the
    // function returns

    if (bResult)
    {
        TelnetWrite(hClient,
                    (LPBYTE)lpszCommand,
                    lstrlen(lpszCommand));

        bResult = TelnetSearch(hClient,
                               _T("$ "),
                               &hgblOutput,
                               &cbOutput,
                               0);
    }

    // Write the contents of the output buffer to the
    // standard output stream

    if (bResult)
    {
        LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

        if (lpBuffer)
            fwrite(lpBuffer, 1, cbBuffer, stdout);

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

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TelnetIsBlocking](#), [TelnetIsReadable](#), [TelnetLogin](#), [TelnetRead](#)

TelnetSetLastError Function

```
VOID WINAPI TelnetSetLastError(  
    DWORD dwErrorCode  
);
```

The **TelnetSetLastError** function sets the last error code for the current thread. This function is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or TELNET_ERROR. Those functions which call **TelnetSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **TelnetGetLastError** function. The use of **TelnetGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstntv10.lib

See Also

[TelnetGetErrorString](#), [TelnetGetLastError](#)

TelnetSetMode Function

```
UINT WINAPI TelnetSetMode(  
    HCLIENT hClient,  
    UINT nMode,  
    BOOL bEnable  
);
```

The **TelnetSetMode** function sets one or more client modes for the specified session.

Parameters

hClient

Handle to the client session.

nMode

The client mode. This value is a combination of one or more flags which determines how the client handles local character echo and character processing. The following values are recognized:

Value	Description
TELNET_MODE_LOCALECHO	The local client is responsible for echoing data entered by the user. By default, this mode is not set which means that the server is responsible for echoing back each character written to it.
TELNET_MODE_BINARY	Data exchanged between the client and server should not be converted or line buffered. If this option is not specified, the high-bit will be cleared on all characters, and single linefeed characters will be converted to carriage-return/linefeed sequences.

bEnable

This boolean flag specifies if the specified mode is to be enabled or disabled.

Return Value

If the function succeeds, the return value is the previous mode. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[TelnetGetMode](#)

TelnetSetTerminalType Function

```
INT WINAPI TelnetSetTerminalType(  
    HCLIENT hClient,  
    LPCTSTR LpszTermType  
);
```

The **TelnetSetTerminalType** function sets the terminal type for the current client session.

Parameters

hClient

Handle to the client session.

lpszTermType

Points to a string which specifies the terminal type.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TelnetGetTerminalType](#)

TelnetSetTimeout Function

```
INT WINAPI TelnetSetTimeout(  
    HCLIENT hClient,  
    INT nTimeout  
);
```

The **TelnetSetTimeout** function sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

See Also

[TelnetGetTimeout](#)

TelnetUninitialize Function

```
VOID WINAPI TelnetUninitialize();
```

The **TelnetUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **TelnetInitialize** call before it can call any of the other library functions. When it has completed the use of library, the application must call **TelnetUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There must be a call to **TelnetUninitialize** for every successful call to **TelnetInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TelnetDisconnect](#), [TelnetInitialize](#)

TelnetWrite Function

```
INT WINAPI TelnetWrite(  
    HCLIENT hClient,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **TelnetWrite** function sends the specified number of bytes to the server.

Parameters

hClient

Handle to the client session.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is TELNET_ERROR. To get extended error information, call **TelnetGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking clients, the client must wait for the TELNET_EVENT_WRITE asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TelnetRead](#)

TelnetWriteLine Function

```
BOOL WINAPI TelnetWriteLine(  
    HCLIENT hClient,  
    LPCTSTR lpszBuffer,  
    LPINT lpnLength  
);
```

The **TelnetWriteLine** function sends a line of text to the server, terminated by a carriage-return and linefeed.

Parameters

hClient

Handle to the client session.

lpszBuffer

The pointer to a string buffer which contains the data that will be sent to the server. All characters up to, but not including, the terminating null character will be written to the server. The data will always be terminated with a carriage-return and linefeed control character sequence. If this parameter points to an empty string or NULL pointer, then a only a carriage-return and linefeed are written to the server.

lpnLength

A pointer to an integer value which will contain the number of characters written to the server, including the carriage-return and linefeed sequence. If this information is not required, a NULL pointer may be specified.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **TelnetGetLastError**.

Remarks

The **TelnetWriteLine** function writes a line of text to the server and terminates the line with a carriage-return and linefeed control character sequence. Unlike the **TelnetWrite** function which writes arbitrary bytes of data to the server, this function is specifically designed to write a single line of text data from a string.

If the *lpszBuffer* string is terminated with a linefeed (LF) or carriage return (CR) character, it will be automatically converted to a standard CRLF end-of-line sequence. Because the string will be sent with a terminating CRLF sequence, the value returned in the *lpnLength* parameter will typically be larger than the original string length (reflecting the additional CR and LF characters), unless the string was already terminated with CRLF.

There are some limitations when using **TelnetWriteLine**. The function should only be used to send text, never binary data. In particular, the function will discard nulls and append linefeed and carriage return control characters to the data stream. The Unicode version of this function will accept a Unicode string, however this function does not support sending raw Unicode data to the server. Unicode strings will be automatically converted to UTF-8 encoding using the **WideCharToMultiByte** function and then written as a stream of bytes.

This function will force the thread to block until the complete line of text has been written, the write operation times out or the server aborts the connection. If this function is called with asynchronous events enabled, it will automatically switch the client into a blocking mode, send the

data and then restore the client to asynchronous operation. If another network operation is attempted while **TelnetWriteLine** is blocked sending data to the server, an error will occur. It is recommended that this function only be used with blocking (synchronous) connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **TelnetWrite** and **TelnetWriteLine** function calls can be safely intermixed.

Unlike the **TelnetWrite** function, it is possible for data to have been written to the server if the return value is zero. For example, if a timeout occurs while the function is waiting to send more data to the server, it will return zero; however, some data may have already been written prior to the error condition. If this is the case, the *lpnLength* argument will specify the number of characters actually written up to that point.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TelnetIsWritable](#), [TelnetRead](#), [TelnetReadLine](#), [TelnetWrite](#)

Telnet Protocol Data Structures

- INITDATA
- SECURITYCREDENTIALS
- SECURITYINFO

INITDATA Structure

This structure contains information about the SocketTools library when the initialization function returns.

```
typedef struct _INITDATA
{
    DWORD        dwSize;
    DWORD        dwVersionMajor;
    DWORD        dwVersionMinor;
    DWORD        dwVersionBuild;
    DWORD        dwOptions;
    DWORD_PTR    dwReserved1;
    DWORD_PTR    dwReserved2;
    TCHAR        szDescription[128];
} INITDATA, *LPINITDATA;
```

Members

dwSize

Size of this structure. This structure member must be set to the size of the structure prior to calling the initialization function. Failure to do so will cause the function to fail.

dwVersionMajor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the major version number for the library.

dwVersionMinor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the minor version number for the library.

dwVersionBuild

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the build number for the library.

dwOptions

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved1

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved2

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

szDescription

A null terminated string which describes the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

SECURITYINFO Structure

This structure contains information about a secure connection that has been established.

```
typedef struct _SECURITYINFO
{
    DWORD        dwSize;
    DWORD        dwProtocol;
    DWORD        dwCipher;
    DWORD        dwCipherStrength;
    DWORD        dwHash;
    DWORD        dwHashStrength;
    DWORD        dwKeyExchange;
    DWORD        dwCertStatus;
    SYSTEMTIME   stCertIssued;
    SYSTEMTIME   stCertExpires;
    LPCTSTR      lpszCertIssuer;
    LPCTSTR      lpszCertSubject;
    LPCTSTR      lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to **sizeof(SECURITYINFO)** prior to passing the address of this structure to the function. Note that if this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the protocol that was selected to establish the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The

	<p>correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.</p>
SECURITY_PROTOCOL_TLS10	<p>The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS11	<p>The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.</p>
SECURITY_PROTOCOL_TLS12	<p>The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.</p>

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection. One of the following values will be returned:

Constant	Description
SECURITY_CIPHER_RC2	The RC2 block cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
SECURITY_CIPHER_DES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher, using 56-bit

	keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher (also known as the Rijndael cipher) is a fixed block size cipher which use a key size of 128, 192 or 256 bits. This cipher is supported on Windows XP SP3 and later versions of the operating system.
SECURITY_CIPHER_SKIPJACK	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
SECURITY_CIPHER_BLOWFISH	The Blowfish block cipher was selected. This is a variable key length cipher up to 448 bits, using a 64-bit block size.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected. One of the following values will be returned:

Constant	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected. One of the following values will be returned:

--	--

Constant	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography. This key exchange algorithm is only supported on Windows XP SP3 and later versions of the operating system.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server. This member only has meaning for connections using the SSL or TLS protocols. One of the following values will be returned:

Constant	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertIssuer

A pointer to a string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszCertSubject

A pointer to a string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL. This member only has meaning for connections using the SSL or TLS protocols.

lpszFingerprint

A pointer to a string which contains a sequence of hexadecimal values that uniquely identify the server. This member is only used when a connection has been established using the Secure Shell (SSH) protocol.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Unicode: Implemented as Unicode and ANSI versions.

SECURITYCREDENTIALS Structure

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate or private key, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Constant	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Both SSL 2.0 and 3.0 have been deprecated and will never be used unless explicitly specified. It is not recommended that you enable either of these protocols for secure connections because it will cause most servers to reject your connection request, even if a version of TLS is also specified.
SECURITY_PROTOCOL_TLS	Either the TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on

	what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers that reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This is the recommended version of TLS to use with secure connections.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Constant	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that

will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A pointer to a null terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A pointer to a null terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A pointer to a null terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A pointer to a null terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
------------	-------------

CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as VeriSign and Thawte act as certification authorities.
----	---

MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
----	--

ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as VeriSign and Thawte are installed as part of the operating system and periodically updated by Microsoft.
------	--

lpszCertName

A pointer to a null terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

A pointer to a null terminated string which specifies the name of the file which contains the private key required to establish the connection. This member is only used for SSH connections

and should always be NULL when establishing a secure connection using SSL or TLS.

Remarks

A client application only needs to create this structure if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* member should be set to CREDENTIAL_STORE_FILENAME and the *lpzCertStore* member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS#12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the *lpzPassword* member or the library will be unable to access the certificate.

Note that the *lpzUserName* and *lpzPassword* members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

The TLS 1.1 and TLS 1.2 protocols are only supported on Windows 7, Windows Server 2008 R2 and later versions of the platform. If these options are specified and the application is running on Windows XP or Windows Vista, the protocol version will be downgraded to TLS 1.0 for backwards compatibility with those platforms.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

Terminal Emulation Library

Emulate an ANSI or DEC VT-220 character mode display terminal.

Reference

- [Functions](#)
- [Data Structures](#)
- [Control Sequences](#)
- [Error Codes](#)

Library Information

File Name	CSNVTV10.DLL
Version	10.0.1468.2518
LibID	53152C92-4EA6-4C06-9ED3-50C79C933F01
Import Library	CSNVTV10.LIB
Dependencies	None

Overview

The Terminal Emulation library provides a comprehensive API for emulating an ANSI or DEC-VT220 terminal, with full support for all standard escape and control sequences, color mapping and other advanced features. The library functions provide both a high level interface for parsing escape sequences and updating a display, as well as lower level primitives for directly managing the virtual display, such as controlling the individual display cells, moving the cursor position and specifying display attributes.

This library can be used in conjunction with the Remote Command, Secure Shell or Telnet Protocol libraries to provide terminal emulation services for an application, or it can be used independently. For example, this library could be used to provide emulation services for a program that connects to a device using an RS-232 serial port.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location

on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Terminal Emulation Functions

Function	Description
NvtClearDisplay	Clear the specified display
NvtConvertDisplayPos	Convert between pixel and screen coordinates
NvtCopySelectedText	Copy the selected text to the clipboard
NvtCreateDisplay	Create a new virtual terminal emulation display
NvtDeleteChar	Delete the specified number of characters
NvtDeleteLine	Delete the current line, shifting remaining lines up
NvtDestroyDisplay	Destroy the specified virtual display
NvtEraseChar	Erase the specified number of characters
NvtEraseLine	Erase the current line
NvtGetCursorPos	Return the current cursor position
NvtGetDisplayAttributes	Return the current display attributes
NvtGetDisplayCell	Return information about the specified cell in the virtual display
NvtGetDisplayCellSize	Return the size of the display character cells
NvtGetDisplayColor	Get the current virtual display colors
NvtGetDisplayColorMap	Return the virtual display color table
NvtGetDisplayDC	Get the current virtual display device context
NvtGetDisplayEmulation	Get the current terminal emulation type
NvtGetDisplayFont	Get the current virtual display font
NvtGetDisplayInfo	Return information about the virtual display
NvtGetDisplayLine	Return a line of text from the virtual display
NvtGetDisplayMode	Get the current virtual display mode
NvtGetDisplayRect	Return the rectangle for the display window client area
NvtGetDisplayScrollPos	Get the display scroll box position
NvtGetDisplaySize	Return the current size of the virtual display
NvtGetDisplayText	Get the specified block of text from the virtual display
NvtGetDisplayWindow	Get the current virtual display window
NvtGetMappedKey	Return the escape sequence for the mapped key
NvtGetScrollRegion	Return the current scrolling region
NvtGetSelectedText	Return the currently selected text
NvtGetTextColor	Return the current text foreground or background color
NvtInitialize	Initialize the library for use by the client

NvtInsertChar	Insert the specified number of characters
NvtInsertLine	Insert a line, shifting the remaining lines down
NvtRefreshDisplay	Refresh the specified display
NvtResetDisplay	Reset the virtual display
NvtResetMappedKeys	Reset the mapped key table to default values
NvtResizeDisplay	Resize the virtual display
NvtRestoreCursor	Restore the saved cursor position and text attributes
NvtSaveCursor	Save the current cursor position and text attributes
NvtScrollDisplay	Scroll the virtual display
NvtSelectDisplayText	Select a region of the virtual display
NvtSetCursorPos	Set the current cursor position
NvtSetDisplayAttributes	Set the current display attributes
NvtSetDisplayBackColor	Set the background color for the virtual display
NvtSetDisplayBoldColor	Set the bold color for the virtual display
NvtSetDisplayCell	Set the value of a character cell in the virtual display
NvtSetDisplayColor	Set the virtual display colors
NvtSetDisplayColorMap	Set the virtual display color table
NvtSetDisplayDC	Set the current virtual display device context
NvtSetDisplayEmulation	Set the current terminal emulation type
NvtSetDisplayFocus	Set the focus on the virtual display
NvtSetDisplayFont	Set the current virtual display font
NvtSetDisplayFontName	Set the current virtual display font by name
NvtSetDisplayForeColor	Set the foreground color for the virtual display
NvtSetDisplayMode	Set the current virtual display mode
NvtSetDisplayScrollPos	Set the display scroll box position
NvtSetDisplaySize	Set the size of the virtual display
NvtSetDisplayWindow	Set the current virtual display window
NvtSetMappedKey	Set the escape sequence for the specified key
NvtSetScrollRegion	Set the current scrolling region
NvtSetTextColor	Set the current text foreground or background color
NvtTranslateMappedKey	Translate the keypress to a mapped key escape sequence
NvtUninitialize	Terminate use of the library by the application
NvtUpdateCaret	Update the display window caret
NvtUpdateDisplay	Update the window attached to the virtual display
NvtWriteDisplay	Write the specified buffer to the virtual display

NvtClearDisplay Function

```
BOOL WINAPI NvtClearDisplay(  
    HDISPLAY hDisplay,  
    UINT nMode  
);
```

The **NvtClearDisplay** function clears the specified display, erasing the text and clearing any attributes.

Parameters

hDisplay

Handle to the virtual display.

nMode

Mode which specifies how the display will be cleared. The following values may be used:

Constant	Description
NVT_CLEAR_EOS	The display is cleared from the current cursor position to the end of the display. The cursor position is not changed.
NVT_CLEAR_TOS	The display is cleared from the beginning of the display to the current cursor position. The cursor position is not changed.
NVT_CLEAR_ALL	The entire display is cleared and the cursor is repositioned to the upper left corner of the display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnavt10.lib

See Also

[NvtDeleteChar](#), [NvtDeleteLine](#), [NvtEraseChar](#), [NvtEraseLine](#), [NvtResetDisplay](#)

NvtConvertDisplayPos Function

```
BOOL WINAPI NvtConvertDisplayPos(  
    HDISPLAY hDisplay,  
    INT nMethod,  
    INT xPos,  
    INT yPos,  
    LPPPOINT lppt  
);
```

The **NvtConvertDisplayPos** function converts the specified X,Y position and stores the result in the POINT structure provided by the caller.

Parameters

hDisplay

Handle to the virtual display.

nMethod

An integer value which specifies the conversion method to use. This may be one of the following values:

Constant	Description
NVT_CURSOR_TO_PIXELS	Convert cursor X,Y coordinates to the current window X,Y pixel coordinates. An error is returned if the coordinates are out of bounds for the current display.
NVT_PIXELS_TO_CURSOR	Convert window X,Y pixel coordinates to cursor X,Y coordinates. If the point is outside of the bounds of the display, it is normalized to account for mouse capture.

xPos

An integer value which specifies the X position in the virtual display. This may either be the cursor position or a pixel position, based on the value of the ***nMethod*** parameter.

yPos

An integer value which specifies the Y position in the virtual display. This may either be the cursor position or a pixel position, based on the value of the ***nMethod*** parameter.

lppt

A pointer to a POINT structure which will contain the converted coordinates.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnavt10.lib

See Also

[NvtGetCursorPos](#), [NvtGetDisplayCellSize](#), [NvtGetDisplayScrollPos](#), [NvtGetDisplaySize](#), [NvtSetCursorPos](#), [NvtSetDisplayScrollPos](#)

NvtCopySelectedText Function

```
BOOL WINAPI NvtCopySelectedText(  
    HDISPLAY hDisplay  
);
```

The **NvtCopySelectedText** function copies any selected text to the system clipboard.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[NvtGetSelectedText](#), [NvtSelectDisplayText](#)

NvtCreateDisplay Function

```
HDISPLAY WINAPI NvtCreateDisplay(  
    HWND hWnd,  
    HFONT hFont,  
    UINT nEmulation,  
    UINT nColumns,  
    UINT nRows  
);
```

The **NvtCreateDisplay** function creates a new virtual terminal emulation display using the specified window and font.

Parameters

hWnd

Handle to the window that will be used to display the virtual terminal.

hFont

Handle to the font that will be used with the virtual terminal. This parameter may be NULL, in which case a default fixed-width font will be used. If a font is specified, it must be fixed-width, otherwise the virtual cursor positioning will be incorrect in some cases.

nEmulation

Identifies the virtual terminal emulation type. The following emulation types are currently supported.

Constant	Description
NVT_EMULATION_NONE	The virtual display does not emulate any specific terminal type, and does not process escape sequences.
NVT_EMULATION_ANSI	The virtual display processes ANSI escape sequences for screen management and cursor positioning. This emulation also supports escape sequences to control the foreground and background color. The default keymap for ANSI function key escape sequences will be selected.
NVT_EMULATION_VT100	The virtual display processes DEC VT-100 escape sequences for screen management and cursor positioning. The default keymap for a DEC VT-100 terminal will be selected.
NVT_EMULATION_VT220	The virtual display processes DEC VT-220 escape sequences for screen management and cursor positioning. This emulation also supports DEC VT-320 escape sequences to control the foreground and background color. The default keymap for a DEC VT-220 terminal will be selected.

nColumns

The maximum number of columns used by the virtual display. This value must be at least 5, and no greater than 255.

nRows

The maximum number of rows used by the virtual display. This value must be at least 5, and no

greater than 127.

Return Value

If the function succeeds, the return value is a handle to the virtual display. If the function fails, the return value is `INVALID_DISPLAY`.

Remarks

The default colors for the display is a black background and white foreground. If ANSI terminal emulation is selected, bold characters will be displayed on a white background and blue foreground.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csnvtv10.lib`

See Also

[NvtDestroyDisplay](#), [NvtInitialize](#), [NvtUninitialize](#)

NvtDeleteChar Function

```
BOOL WINAPI NvtDeleteChar(  
    HDISPLAY hDisplay,  
    INT nChars  
);
```

The **NvtDeleteChar** function deletes the specified number of characters from the display, shifting the characters that follow to the left. The characters are deleted from the current cursor position.

Parameters

hDisplay

Handle to the virtual display.

nChars

Number of characters to delete from the display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function does not change the current cursor position. To erase characters from the display without affecting the characters that follow, use the **NvtEraseChar** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[NvtClearDisplay](#), [NvtDeleteLine](#), [NvtEraseChar](#), [NvtEraseLine](#)

NvtDeleteLine Function

```
BOOL WINAPI NvtDeleteLine(  
    HDISPLAY hDisplay  
);
```

The **NvtDeleteLine** function deletes the current line, shifting up the lines that follow.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function does not change the current cursor position. To erase a line from the display without affecting the lines that follow, use the **NvtEraseLine** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnvtv10.lib

See Also

[NvtClearDisplay](#), [NvtDeleteChar](#), [NvtEraseChar](#), [NvtEraseLine](#)

NvtDestroyDisplay Function

```
VOID WINAPI NvtDestroyDisplay(  
    HDISPLAY hDisplay  
);
```

The **NvtDestroyDisplay** function releases the memory allocated for the virtual display.

Parameters

hDisplay

Handle to the virtual display.

Return Value

None.

Remarks

If the display font was specified when the display was created, then it is the responsibility of the application to delete the font object using the **DeleteObject** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnavt10.lib

See Also

[NvtCreateDisplay](#), [NvtInitialize](#), [NvtUninitialize](#)

NvtEraseChar Function

```
BOOL WINAPI NvtEraseChar(  
    HDISPLAY hDisplay,  
    INT nChars  
);
```

The **NvtEraseChar** function erases the specified number of characters from the display, without affecting the position of the characters that follow. The characters are erased from the current cursor position.

Parameters

hDisplay

Handle to the virtual display.

nChars

Number of characters to erase from the display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function does not change the current cursor position. To delete characters from the display and shift the remaining characters to the left, use the **NvtDeleteChar** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[NvtClearDisplay](#), [NvtDeleteChar](#), [NvtDeleteLine](#), [NvtEraseLine](#)

NvtEraseLine Function

```
BOOL WINAPI NvtEraseLine(  
    HDISPLAY hDisplay,  
    UINT nMode  
);
```

The **NvtEraseLine** function erases the current line without affecting the lines that follow.

Parameters

hDisplay

Handle to the virtual display.

nMode

Mode which specifies how the line will be erased. The following values may be used:

Value	Description
0	The line is erased from the current cursor position to the end of the line.
1	The line is cleared from the beginning of the line to the current cursor position.
2	The entire line is cleared.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function does not change the current cursor position. To delete a line from the display and shift the remaining lines up, use the **NvtDeleteLine** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csnvtv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NvtClearDisplay](#), [NvtDeleteChar](#), [NvtDeleteLine](#), [NvtEraseChar](#)

NvtGetCursorPos Function

```
BOOL WINAPI NvtGetCursorPos(  
    HDISPLAY hDisplay,  
    LPINT lpnCursorX,  
    LPINT lpnCursorY  
);
```

The **NvtGetCursorPos** function returns the current cursor column and row position on the virtual display.

Parameters

hDisplay

Handle to the virtual display.

lpnCursorX

Address of the variable that will be set to the column of the current cursor position. If this argument is a NULL pointer, the argument is ignored.

lpnCursorY

Address of the variable that will be set to the row of the current cursor position. If this argument is a NULL pointer, the argument is ignored.

Return Value

If the function succeeds, the return value is a non-zero value. If the handle to the virtual display is invalid, the function will return zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[NvtConvertDisplayPos](#), [NvtSetCursorPos](#)

NvtGetDisplayAttributes Function

```
UINT WINAPI NvtGetDisplayAttributes(  
    HDISPLAY hDisplay  
);
```

The **NvtGetDisplayAttributes** function returns the current display attributes which have been set, either explicitly by the client or as the result of an escape sequence parsed by the emulator.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is the current display attributes. If the function fails, the return value is NVT_ERROR.

The following table lists the valid attributes:

Constant	Description
NVT_ATTRIBUTE_NORMAL	Normal, default attributes.
NVT_ATTRIBUTE_REVERSE	Foreground and background cell colors are reversed.
NVT_ATTRIBUTE_BOLD	The character is displayed using a higher intensity color.
NVT_ATTRIBUTE_DIM	The character is displayed using a lower intensity color.
NVT_ATTRIBUTE_UNDERLINE	The character is displayed with an underline.
NVT_ATTRIBUTE_HIDDEN	The character is stored in display memory, but not shown.
NVT_ATTRIBUTE_PROTECT	The character is protected and cannot be cleared.

One or more attributes may be combined using a bitwise Or operator. Certain attributes, such as NVT_ATTRIBUTE_BOLD and NVT_ATTRIBUTE_DIM are mutually exclusive.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[NvtGetDisplayCell](#), [NvtSetDisplayAttributes](#)

NvtGetDisplayCell Function

```
DWORD WINAPI NvtGetDisplayCell(  
    HDISPLAY hDisplay,  
    INT xPos,  
    INT yPos  
);
```

The **NvtGetDisplayCell** function returns information about the specified character cell in the display.

Parameters

hDisplay

Handle to the virtual display.

xPos

An integer value which specifies the X cursor position in the display.

yPos

An integer value which specifies the Y cursor position in the display.

Return Value

If the function succeeds, the return value is the cell character and attributes. If the function fails, it returns NVT_ERROR.

Remarks

The value returned by the **NvtGetDisplayCell** function is an unsigned 32-bit integer value, where the low order word specifies the ANSI character stored at that position and the high order word specifies the display attributes for that cell.

The character cell attributes may be one or more of the following values:

Constant	Description
NVT_ATTRIBUTE_NORMAL	Normal, default attributes.
NVT_ATTRIBUTE_REVERSE	Foreground and background cell colors are reversed.
NVT_ATTRIBUTE_BOLD	The character is displayed using a higher intensity color.
NVT_ATTRIBUTE_DIM	The character is displayed using a lower intensity color.
NVT_ATTRIBUTE_UNDERLINE	The character is displayed with an underline.
NVT_ATTRIBUTE_HIDDEN	The character is stored in display memory, but not shown.
NVT_ATTRIBUTE_PROTECT	The character is protected and cannot be cleared.

One or more attributes may be combined using a bitwise Or operator. Certain attributes, such as NVT_ATTRIBUTE_BOLD and NVT_ATTRIBUTE_DIM are mutually exclusive.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnvtv10.lib

See Also

[NvtGetDisplayAttributes](#), [NvtGetDisplayCellSize](#), [NvtSetDisplayAttributes](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

NvtGetDisplayCellSize Function

```
BOOL WINAPI NvtGetDisplayCellSize(  
    HDISPLAY hDisplay,  
    LPSIZE lpCellSize  
);
```

The **NvtGetDisplayCellSize** function returns the size of a character cell in pixels.

Parameters

hDisplay

Handle to the virtual display.

lpCellSize

A pointer to a SIZE structure which will contain the size of a character cell in pixels.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

The **NvtGetDisplayCellSize** function is used to determine the size of a character cell in the display. This can be useful when the application needs to determine where a display cell is physically located within the virtual display window.

To convert between display and window coordinates, use the **NvtConvertDisplayPos** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[NvtConvertDisplayPos](#), [NvtGetDisplayCell](#)

NvtGetDisplayColor Function

```
COLORREF WINAPI NvtGetDisplayColor(  
    HDISPLAY hDisplay,  
    UINT nColorIndex,  
    BOOL bForeground  
);
```

The **NvtGetDisplayColor** function returns the color used by the virtual display to display text with a specific attribute.

Parameters

hDisplay

Handle to the virtual display.

nColorIndex

The index into the virtual display color table. It may be one of the following values.

Value	Description
NVT_COLOR_NORMAL	The colors displayed for normal text. These are the default colors used with the display.
NVT_COLOR_REVERSE	The colors displayed for text with the reverse attribute set. This is only used when emulation is enabled.
NVT_COLOR_BOLD	The colors displayed for text with the bold attribute set. This is only used when emulation is enabled.
NVT_COLOR_REVERSEBOLD	The colors displayed for text with the reverse and bold attributes set. This is only used when emulation is enabled.

bForeground

A boolean flag which specifies if the color is a foreground color, used when displaying text, or a background color.

Return Value

If the function succeeds, the return value is the RGB value of the specified color. If the function fails, it returns zero.

Remarks

The default colors for the display is a black background and white foreground. Bold characters are displayed on a white background and blue foreground.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csntv10.lib

See Also

[NvtSetDisplayBackColor](#), [NvtSetDisplayBoldColor](#), [NvtSetDisplayColor](#), [NvtSetDisplayForeColor](#)

NvtGetDisplayColorMap Function

```
BOOL WINAPI NvtGetDisplayColorMap(  
    HDISPLAY hDisplay,  
    COLORREF* lpColor,  
    INT nMaxColors  
);
```

The **NvtGetDisplayColorMap** function returns the virtual display color table which determines what RGB values are used to display foreground and background text color attributes.

Parameters

hDisplay

Handle to the virtual display.

lpColor

A pointer to an array of COLORREF values which will contain the color values currently being used in the virtual display.

nMaxColors

The maximum number of colors which may be stored in the color array. The minimum value for this parameter is 1, and the maximum value is 16.

Return Value

If the function succeeds, the return value is a non-zero value. If the function fails, the return value is zero. Failure indicates that the handle to the virtual display is invalid, the pointer to the color table is NULL or the maximum number of color values is invalid.

Remarks

When the emulator processes an escape sequence that changes the current foreground or background color, the actual RGB color value is determined by looking up the value in the virtual display's color table. The **NvtGetDisplayColorMap** function is useful for determining what values are being used when a color attribute is set. The emulator currently supports a maximum of sixteen (16) color values, and the index into the table corresponds to the color as defined by the standard for ANSI terminals:

Index	Color	Default (Hex)	Default (Integer)	Default (RGB)
0	Black	0	0	RGB(0,0,0)
1	Red	000000A0h	160	RGB(160,0,0)
2	Green	0000A000h	40960	RGB(0,160,0)
3	Yellow	0000A0A0h	41120	RGB(160,160,0)
4	Blue	00A00000h	10485760	RGB(0,0,160)
5	Magenta	00A000A0h	10485920	RGB(160,0,160)
6	Cyan	00A0A000h	10526720	RGB(0,160,160)
7	White	00E0E0E0h	14737632	RGB(224,224,224)
8	Gray	00C0C0C0h	12632256	RGB(192,192,192)
9	Light Red	008080FFh	8421631	RGB(255,128,128)

10	Light Green	0090EE90h	9498256	RGB(144,238,144)
11	Light Yellow	00C0FFFFh	12648447	RGB(255,255,192)
12	Light Blue	00E6D8ADh	15128749	RGB(173,216,230)
13	Light Magenta	00FFC0FFh	16761087	RGB(255,192,255)
14	Light Cyan	00FFFFFF0h	16777184	RGB(224,255,255)
15	High White	00FFFFFFh	16777215	RGB(255,255,255)

A standard ANSI color terminal supports eight standard colors (0-7). To select a foreground color, you add 30 to the color index and pass that value as a parameter to the SGR (select graphic rendition) escape sequence. To select a background color, you add 40 to the color index. For example, to set the current foreground color to white and the background color to blue, you could send the following escape sequence:

```
ESC [ 37;44 m
```

Note that if you wanted to set the foreground color to a bold version of standard yellow, you would first set the bold attribute, and then use the index value of 3, such as:

```
ESC [ 1;33m
```

NvtGetDisplayColorMap is typically used in conjunction with the **NvtSetDisplayColorMap** function to load the current color values and then make selective changes to the actual RGB color value that is used when a color attribute is set. Note that changes to the color map will only affect new characters as they are displayed, not any previously displayed characters.

Example

The following example will load the current color table for the virtual display and change the standard white color attribute to use the same value as the high-intensity white:

```
COLORREF rgbColor[16];

if ( NvtGetDisplayColorMap(hDisplay, rgbColor, 16) )
{
    rgbColor[7] = rgbColor[15];
    NvtSetDisplayColorMap(hDisplay, rgbColor, 16);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: csnavt10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NvtGetDisplayColor](#), [NvtGetTextColor](#), [NvtSetDisplayColor](#), [NvtSetDisplayColorMap](#), [NvtSetTextColor](#)

NvtGetDisplayDC Function

```
HDC WINAPI NvtGetDisplayDC(  
    HDISPLAY hDisplay  
);
```

The **NvtGetDisplayDC** returns the device context that has been specified for the virtual display.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is a handle to the device context. If the function fails, or no device context has been specified, the return value is NULL.

Remarks

Normally there is no device context explicitly set for the display. Instead, the device context is dynamically created and released when the virtual display is updated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[NvtSetDisplayDC](#), [NvtUpdateDisplay](#)

NvtGetDisplayEmulation Function

```
UINT WINAPI NvtGetDisplayEmulation(  
    HDISPLAY hDisplay  
);
```

The **NvtGetDisplayEmulation** function returns the current virtual terminal emulation type.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is the terminal emulation type and may contain one of the following values:

Value	Description
NVT_EMULATION_NONE	The virtual display does not emulate any specific terminal type, and does not process any escape sequences.
NVT_EMULATION_ANSI	The virtual display will process ANSI escape sequences. The default keymap for an ANSI console is loaded.
NVT_EMULATION_VT100	The virtual display will process DEC VT100 escape sequences. The default keymap for a VT100 terminal is loaded.
NVT_EMULATION_VT220	The virtual display will process DEC VT220 escape sequences. The default keymap for a VT220 terminal is loaded.

If the function fails, because an invalid display handle was specified, the return value will be NVT_ERROR.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnavt10.lib

See Also

[NvtCreateDisplay](#), [NvtResetDisplay](#), [NvtSetDisplayEmulation](#)

NvtGetDisplayFont Function

```
HFONT WINAPI NvtGetDisplayFont(  
    HDISPLAY hDisplay  
);
```

The **NvtGetDisplayFont** function returns the current font handle being used by the virtual display.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is a handle to the font. If the function fails, it returns NULL.

Remarks

The handle returned by this function should not be released, and the font object should never be directly modified by the application. Functions that return information about the font, such as **GetTextMetrics**, may be safely called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: csnavt10.lib

See Also

[NvtCreateDisplay](#), [NvtSetDisplayFont](#), [NvtSetDisplayFontName](#)

NvtGetDisplayInfo Function

```
BOOL WINAPI NvtGetDisplayInfo(  
    HDISPLAY hDisplay,  
    LPNVTDISPLAYINFO lpvdi  
);
```

The **NvtGetDisplayInfo** function returns information about the specified virtual display.

Parameters

hDisplay

Handle to the virtual display.

lpvdi

Pointer to a [NVTDISPLAYINFO](#) structure which contains information about the virtual display, including the display window, font, cursor and scrolling position.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[NvtGetCursorPos](#), [NvtGetDisplayFont](#), [NvtGetDisplayMode](#), [NvtGetDisplayRect](#),
[NvtGetDisplayScrollPos](#)

NvtGetDisplayLine Function

```
INT WINAPI NvtGetDisplayLine(  
    HDISPLAY hDisplay,  
    INT nRow,  
    LPTSTR LpszBuffer,  
    INT nMaxLength  
);
```

The **NvtGetDisplayLine** function copies a block of text from the specified virtual display into a string buffer.

Parameters

hDisplay

Handle to the virtual display.

nRow

The row in the virtual display to return the line of text from. The first row in the display is zero. If the value -1 is specified, the row where the cursor is currently located will be used.

lpszBuffer

Pointer to the buffer that the display text will be copied to, terminated with a null character character.

cbBuffer

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the function succeeds, the return value is the number of characters copied into the buffer, not including the null character terminator. If the function fails, it returns zero.

Remarks

The **NvtGetDisplayLine** function allows the application to copy the contents of the display at a specific row. Note that the buffer must be large enough to accommodate the text and the null character terminator. Unlike the **NvtGetDisplayText** function, any trailing whitespace in the specified row is not copied to the buffer.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NvtGetDisplayText](#), [NvtUpdateDisplay](#), [NvtWriteDisplay](#)

NvtGetDisplayMode Function

```
UINT WINAPI NvtGetDisplayMode(  
    HDISPLAY hDisplay  
);
```

The **NvtGetDisplayMode** function returns the current virtual display modes.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is the display mode for the virtual display. If the function fails, it returns zero.

Remarks

The display mode is a combination of one or more flags which determines how the emulator handles automatic line wrapping, caret display, and other functions. The following values are recognized.

Constant	Description
NVT_MODE_AUTOWRAP	The emulator will automatically wrap to the next line when a character is written to the last column on the virtual display.
NVT_MODE_SHOWCARET	The emulator will display a caret when the display receives the focus.
NVT_MODE_BLOCKCARET	The emulator will display a block carat that is the height of the selected font characters. If this mode is not set, the caret is displayed as an underline.
NVT_MODE_BELL	The emulator will beep when a BEL character is processed.
NVT_MODE_CRLF	The cursor will automatically be positioned at the first column when a linefeed character is processed.
NVT_MODE_CRNL	The cursor will automatically advance to the next row when a carriage return character is processed.
NVT_MODE_APPCURSOR	The emulator cursor keys are placed in application mode. This mode changes the default key mappings used when the cursor (arrow) keys are translated. This corresponds to the application mode supported by DEC VT terminals.
NVT_MODE_APPKEYPAD	The emulator keypad keys are placed in application mode. This mode changes the default key mappings used when the keypad keys are translated. This corresponds to the application keypad mode supported by DEC VT terminals.
NVT_MODE_ORIGIN	The emulator is in origin mode. If enabled, the cursor cannot be positioned outside of the current scrolling region. Otherwise, the cursor can be positioned at any valid location on the virtual display.
NVT_MODE_COLOR	The emulator supports the use of escape sequences to change the

	foreground and background colors. This option is enabled by default if emulating an ANSI console or DEC VT220 terminal.
NVT_MODE_TABOVER	The emulator will clear the character cells between the current cursor position and the next tab stop when the HT (horizontal tab) control sequence is processed. By default this mode is disabled, and the cursor is simply positioned at the next tab stop.
NVT_MODE_HSCROLL	The emulator will display a horizontal scroll bar if the number of visible columns are less than that total number of columns in the virtual display. Disabling this mode prevents a horizontal scrollbar from being displayed, regardless of the number of visible columns. By default, this mode is enabled.
NVT_MODE_VSCROLL	The emulator will display a vertical scroll bar if the number of visible rows are less than that total number of rows in the virtual display. Disabling this mode prevents a vertical scrollbar from being displayed, regardless of the number of visible rows. By default, this mode is enabled.
NVT_MODE_NOREFRESH	The emulator will not automatically refresh the window after any change has been made to the virtual display, including changes in the cursor position or display mode. This allows the caller to make a sequence of changes, and then update the display all at one time to prevent a flicker effect.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnavt10.lib

See Also

[NvtGetDisplayEmulation](#), [NvtGetDisplayInfo](#), [NvtSetDisplayMode](#)

NvtGetDisplayRect Function

```
BOOL WINAPI NvtGetDisplayRect(  
    HDISPLAY hDisplay,  
    LPRECT lprc  
);
```

The **NvtGetDisplayRect** returns the client rectangle for the virtual display window.

Parameters

hDisplay

Handle to the virtual display.

lpRect

Pointer to a RECT structure which will contain the client rectangle values when the function returns.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnavt10.lib

See Also

[NvtGetDisplayInfo](#), [NvtGetDisplaySize](#), [NvtResizeDisplay](#), [NvtSetDisplaySize](#)

NvtGetDisplayScrollPos Function

```
INT WINAPI NvtGetDisplayScrollPos(  
    HDISPLAY hDisplay,  
    INT nScrollBar  
);
```

The **NvtGetDisplayScrollPos** function gets the position of the scroll box for the specified scroll bar.

Parameters

hDisplay

Handle to the virtual display.

nScrollBar

Specifies the scroll bar to return the position for. This parameter can be one of the following values:

Constant	Description
SB_HORZ	Gets the position of the scroll box in the display's standard horizontal scroll bar.
SB_VERT	Gets the position of the scroll box in the display's standard vertical scroll bar.

Return Value

If the function succeeds, the return value is the position of the scroll box. If the function fails, it returns -1.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[NvtSetDisplayScrollPos](#)

NvtGetDisplaySize Function

```
BOOL WINAPI NvtGetDisplaySize(  
    HDISPLAY hDisplay,  
    LPSIZE lpSize  
);
```

The **NvtGetDisplaySize** returns the size of the virtual display in columns and rows.

Parameters

hDisplay

Handle to the virtual display.

lpRect

Pointer to a SIZE structure which will contain the size of the virtual display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

The **NvtGetDisplaySize** function returns the number of columns and rows in the virtual display. To convert this to pixels, use the **NvtGetDisplayCellSize** function to determine the size of a character cell and multiply that by the number of columns and/or rows.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnavt10.lib

See Also

[NvtGetDisplayCellSize](#), [NvtGetDisplayInfo](#), [NvtGetDisplayRect](#)

NvtGetDisplayText Function

```
INT WINAPI NvtGetDisplayText(  
    HDISPLAY hDisplay,  
    INT nOffset,  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);
```

The **NvtGetDisplayText** function copies a block of text from the specified virtual display into a string buffer.

Parameters

hDisplay

Handle to the virtual display.

nOffset

Offset into the virtual display buffer. A value of -1 specifies that the current cursor position should be used as the offset.

lpszBuffer

Pointer to the buffer that the display text will be copied to, terminated with a null character character.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the function succeeds, the return value is the number of characters copied into the buffer, not including the null character terminator. If the function fails, it returns zero.

Remarks

The **NvtGetDisplayText** function allows the application to copy the contents of the display at a specific location. Note that the buffer must be large enough to accommodate the text and the null character terminator. To copy an entire row of text in the display, use the **NvtGetDisplayLine** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnavt10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NvtGetDisplayLine](#), [NvtUpdateDisplay](#), [NvtWriteDisplay](#)

NvtGetDisplayWindow Function

```
HWND WINAPI NvtGetDisplayWindow(  
    HDISPLAY hDisplay  
);
```

The **NvtGetDisplayWindow** returns the handle to the window being used by the virtual display.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is a handle to the virtual display window. If the function fails, it returns NULL.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnavt10.lib

See Also

[NvtGetDisplayInfo](#)

NvtGetMappedKey Function

```
BOOL WINAPI NvtGetMappedKey(  
    HDISPLAY hDisplay,  
    UINT nMappedKey,  
    LPTSTR lpszKeyBuffer,  
    UINT cchKeyBuffer  
);
```

The **NvtGetMappedKey** function returns the escape sequence mapped to the specified key.

Parameters

hDisplay

Handle to the virtual display.

nMappedKey

The key which is an index into the display key mapping table. This defines the values returned by special (function) keys for the current emulation.

lpszKeyBuffer

Address of the buffer which will receive the escape sequence mapped to the specified key.

cchKeyBuffer

The maximum number of characters that may be copied into the key buffer string, including the terminating null character.

Return Value

If the function succeeds, the return value is non-zero and the escape sequence for the mapped key is copied into the specified buffer. If the key has not been mapped, and there is no default escape sequence defined, then the function will return zero.

Remarks

The **NvtGetMappedKey** function returns the escape sequence that has been mapped to a special key. This function can be used to determine what sequence of characters should be sent in response to a keypress (for example, what sequence should be sent to a server when the user presses the F1 function key). If a sequence has not been explicitly mapped through a call to **NvtSetMappedKey**, the default sequence for the current emulation will be returned.

Note that the current display mode, such as whether or not the emulator is in application mode or not, should be considered when determining which mapped key to use. For example, if the emulator is not in application mode and the user presses the up-arrow key, the sequence mapped to the NVT_UP key should be sent to the server. However, if the emulator is in application mode, the sequence mapped to the NVT_APPUP key should be sent instead. This is automatically handled by the **NvtTranslateMappedKey** function, so it is recommended that it be used when mapping a virtual keypress to the appropriate escape sequence.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnavt10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NvtGetDisplayMode](#), [NvtResetMappedKeys](#), [NvtSetDisplayMode](#), [NvtSetMappedKey](#),
[NvtTranslateMappedKey](#)

NvtGetScrollRegion Function

```
BOOL WINAPI NvtGetScrollRegion(  
    HDISPLAY hDisplay,  
    LPINT lpnTop,  
    LPINT lpnBottom  
);
```

The **NvtGetScrollRegion** function returns the top and bottom rows of the current scrolling region.

Parameters

hDisplay

Handle to the virtual display.

lpnTop

Address of the variable that will be set to the top row of the current scrolling region. If no scrolling region has been defined, the value will be 0, the first row in the virtual display. If a NULL pointer is passed as the value, this argument will be ignored.

lpnBottom

Address of the variable that will be set to the bottom row of the current scrolling region. If no scrolling region has been defined, the value will be one less than the maximum number of rows in the virtual display. If a NULL pointer is passed as the value, this argument will be ignored.

Return Value

If the function succeeds, it will return zero. If the handle to the display is invalid, the function will return zero.

Remarks

The **NvtGetScrollRegion** function allows an application to determine the top and bottom rows of the current scrolling region. By default, the scrolling region is the entire virtual display, however this may be changed through a call to **NvtSetScrollRegion** or an ANSI escape sequence. If the display is in origin mode, note that the cursor cannot be positioned outside of the current scrolling region.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnavt10.lib

See Also

[NvtGetDisplayMode](#), [NvtSetDisplayMode](#), [NvtSetScrollRegion](#)

NvtGetSelectedText Function

```
INT WINAPI NvtGetSelectedText(  
    HDISPLAY hDisplay,  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);
```

The **NvtGetSelectedText** function copies the currently selected text into the specified buffer.

Parameters

hDisplay

Handle to the virtual display.

lpszBuffer

Pointer to the buffer that the selected text will be copied to, terminated with a null character character.

nMaxLength

Maximum number of characters that may be copied into the specified buffer, including the null character terminator.

Return Value

If the function succeeds, the return value is the number of characters copied into the buffer, not including the null character terminator. If the function fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NvtCopySelectedText](#), [NvtSelectDisplayText](#)

NvtGetTextColor Function

```
BOOL WINAPI NvtGetTextColor(  
    HDISPLAY hDisplay,  
    COLORREF *lprgbColor,  
    BOOL bForeground  
);
```

The **NvtGetTextColor** function returns the current foreground or background text color.

Parameters

hDisplay

Handle to the virtual display.

lprgbColor

A pointer to a COLORREF variable which will contain the current foreground or background text color.

bForeground

A boolean value which determines if the foreground or background color is returned. A value of TRUE indicates that the foreground color should be returned, otherwise the background color is returned.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. Failure indicates that the handle to the display is invalid, the display does not support text color attributes, or the pointer to the color value is NULL.

Remarks

This function is used to return the current foreground or background color, as determined by the text attribute. The RGB color value for a color attribute is determined by the virtual display's color table.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[NvtGetDisplayColorMap](#), [NvtSetDisplayColorMap](#), [NvtSetTextColor](#)

NvtInitialize Function

```
BOOL WINAPI NvtInitialize(  
    LPCTSTR lpszLicenseKey,  
    LPINITDATA lpData  
);
```

The **NvtInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpData

Pointer to an INITDATA data structure. This parameter may be NULL if the initialization data for the library is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **NvtGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must insure that this function is called with a valid license key.

If the *lpData* argument is specified, it must point to an INITDATA structure which has been initialized by setting the *dwSize* member to the size of the structure. All other structure members should be set to zero. If the function is successful, then the INITDATA structure will be filled with identifying information about the library.

Although it is only required that **NvtInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **NvtUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **NvtInitialize** or **NvtUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

NvtCreateDisplay, NvtDestroyDisplay, NvtResetDisplay, NvtUninitialize

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

NvtInsertChar Function

```
BOOL WINAPI NvtInsertChar(  
    HDISPLAY hDisplay,  
    INT nChars  
);
```

The **NvtInsertChar** function inserts one or more space characters at the current cursor position.

Parameters

hDisplay

Handle to the virtual display.

nChars

Number of space characters to insert at the current cursor position.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function does not change the cursor position. Inserting space characters may cause the virtual display to scroll.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csntv10.lib

See Also

[NvtDeleteChar](#), [NvtEraseChar](#), [NvtInsertLine](#)

NvtInsertLine Function

```
BOOL WINAPI NvtInsertLine(  
    HDISPLAY hDisplay  
);
```

The **NvtInsertLine** function inserts an empty line at the current row, shifting the remaining lines down.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function does not change the cursor position. Inserting a line may cause the virtual display to scroll.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[NvtDeleteLine](#), [NvtEraseLine](#), [NvtInsertChar](#)

NvtRefreshDisplay Function

```
BOOL WINAPI NvtRefreshDisplay(  
    HDISPLAY hDisplay,  
    BOOL bUpdate  
);
```

The **NvtRefreshDisplay** function refreshes the specified virtual display, updating the current scroll position and caret.

Parameters

hDisplay

Handle to the virtual display.

bUpdate

Boolean flags which specifies if the display window is to be updated. If set, the window client area is invalidated and the virtual display is redrawn.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnvtv10.lib

See Also

[NvtUpdateCaret](#), [NvtUpdateDisplay](#), [NvtWriteDisplay](#)

NvtResetDisplay Function

```
BOOL WINAPI NvtResetDisplay(  
    HDISPLAY hDisplay,  
    HWND hWnd,  
    HFONT hFont,  
    UINT nColumns,  
    UINT nRows  
);
```

The **NvtResetDisplay** resets the virtual terminal display, using the new window, font, columns and rows. This function should be used when the virtual display must be attached to a different window, or the number of rows or columns must be changed.

Parameters

hDisplay

Handle to the virtual display.

hWnd

Handle to the window that will be used to display the virtual terminal. This parameter may be NULL, in which case the current window will be used.

hFont

Handle to the font that will be used with the virtual terminal. This parameter may be NULL, in which case the current font will be used. If a font is specified, it must be fixed-width, otherwise the virtual cursor positioning will be incorrect in some cases.

nColumns

The maximum number of columns used by the virtual display. A value of zero specifies that the same number of columns should be used. If a non-zero value is specified, it must be at least 5, and no greater than 255.

nRows

The maximum number of rows used by the virtual display. A value of zero specifies that the same number of columns should be used. If a non-zero value is specified, it must be at least 5, and no greater than 127.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Remarks

This function will clear the display and reset the current text attributes.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[NvtClearDisplay](#), [NvtCreateDisplay](#), [NvtRefreshDisplay](#), [NvtResizeDisplay](#), [NvtUpdateDisplay](#)

NvtResetMappedKeys Function

```
BOOL WINAPI NvtResetMappedKeys(  
    HDISPLAY hDisplay  
);
```

The **NvtResetMappedKeys** function resets all mapped function keys to their default values, based on the current emulation.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is non-zero. If the handle to the display is invalid, the function will return zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[NvtGetMappedKey](#), [NvtSetMappedKey](#), [NvtTranslateMappedKey](#)

NvtResizeDisplay Function

```
BOOL WINAPI NvtResizeDisplay(  
    HDISPLAY hDisplay,  
    INT cxClient,  
    INT cyClient  
);
```

The **NvtResizeDisplay** function resizes the virtual display to the specified width and height.

Parameters

hDisplay

Handle to the virtual display.

cxClient

New width of the display window in pixels. If this value is zero, the width of the virtual display remains unchanged.

cyClient

New height of the display window in pixels. If this value is zero, the height of the virtual display remains unchanged.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function resizes the virtual display and updates the scrolling information. Typically this function is called when the display window receives a WM_SIZE message, causing the virtual display to match the size of the window's client area.

This function will not change the size of the display window. To change the size of the display window, use the **SetWindowPos** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NvtCreateDisplay](#), [NvtGetDisplayRect](#), [NvtRefreshDisplay](#), [NvtResetDisplay](#), [NvtUpdateDisplay](#)

NvtRestoreCursor Function

```
BOOL WINAPI NvtRestoreCursor(  
    HDISPLAY hDisplay  
);
```

The **NvtRestoreCursor** function restores the cursor position and text attributes to their previous values.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

Use the **NvtSaveCursor** function to save the current cursor position and text attributes. This function may only be called once for each time that the cursor position is saved.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[NvtGetCursorPos](#), [NvtSaveCursor](#), [NvtSetCursorPos](#), [NvtUpdateCaret](#), [NvtUpdateDisplay](#)

NvtSaveCursor Function

```
BOOL WINAPI NvtSaveCursor(  
    HDISPLAY hDisplay  
);
```

The **NvtSaveCursor** function saves the current cursor position and text attributes.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

Use the **NvtRestoreCursor** function to restore the cursor position and text attributes to their previous values.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[NvtGetCursorPos](#), [NvtRestoreCursor](#), [NvtSetCursorPos](#), [NvtUpdateCaret](#), [NvtUpdateDisplay](#)

NvtScrollDisplay Function

```
BOOL WINAPI NvtScrollDisplay(  
    HDISPLAY hDisplay,  
    BOOL bScrollUp  
);
```

The **NvtScrollDisplay** function scrolls the virtual display up or down and updates the scroll box position if necessary.

Parameters

hDisplay

Handle to the virtual display.

bScrollUp

Boolean flag which specifies if the virtual display is scrolled up or down.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function does not change the current cursor position. The **NvtRefreshDisplay** function should be called to update the window attached to the virtual display.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnavt10.lib

See Also

[NvtGetDisplayScrollPos](#), [NvtRefreshDisplay](#), [NvtSetDisplayScrollPos](#)

NvtSelectDisplayText Function

```
BOOL WINAPI NvtSelectDisplayText(  
    HDISPLAY hDisplay,  
    LPRECT lprc,  
    DWORD dwOptions  
);
```

The **NvtSelectDisplayText** function selects or unselects a region of the virtual display.

Parameters

hDisplay

Handle to the virtual display.

lprc

A pointer to a region of the display to select. The coordinates must be in display cursor coordinates, not pixels. If this parameter is NULL, any selected text in the display is unselected.

dwOptions

One or more options which specifies how the region will be selected. These options may be combined using a bitwise Or operator. The following values may be used:

Constant	Description
NVT_SELECT_DEFAULT	The default selection option. If there is a region of the display already selected, it will be cleared and the new region is selected.
NVT_SELECT_CLIPBOARD	Copy the selected text to the clipboard. If this option is not specified, the selected text is buffered and may be copied at a later point.
NVT_SELECT_NOREFRESH	The display is not refreshed when the region is selected. This is useful if the application is going to be selecting multiple regions of the display, or combining more than one region, in order to minimize output to the window.
NVT_SELECT_NOBUFFER	Do not buffer the text in the selected region of the display. The display will show any text as being selected, but it will not be available to be copied by the application. This can be useful if the application is going to select multiple regions and combine them.
NVT_SELECT_COMBINE	If there is already a region of the display that has been selected, the new region is combined with the previous region, selecting all of the text.
NVT_SELECT_UNSELECT	Unselect the specified region of the display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnavt10.lib

See Also

[NvtCopySelectedText](#), [NvtGetSelectedText](#)

NvtSetCursorPos Function

```
BOOL WINAPI NvtSetCursorPos(  
    HDISPLAY hDisplay,  
    INT nCursorX,  
    INT nCursorY  
);
```

The **NvtSetCursorPos** function sets the current cursor column and row position on the virtual display.

Parameters

hDisplay

Handle to the virtual display.

nCursorX

New cursor column position. If this value is greater than the maximum number of columns, the current position is set to the last column on the display. The first column on the display is zero.

nCursorY

New cursor row position. If this value is greater than the maximum number of rows, the current position is set to the last row on the display. The first row on the display is zero.

Return Value

If the function succeeds, the return value is non-zero. If the handle to the virtual display is invalid, the function will return zero.

Remarks

The **NvtSetCursorPos** function sets the current cursor position on the virtual display. If the display is in origin mode (a scrolling region has been set), then the cursor row position is bound by the current region.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `csnvtv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NvtGetCursorPos](#)

NvtSetDisplayAttributes Function

```
UINT WINAPI NvtSetDisplayAttributes(  
    HDISPLAY hDisplay,  
    UINT nAttributes  
);
```

The **NvtSetDisplayAttributes** function sets the current display attributes.

Parameters

hDisplay

Handle to the virtual display.

nAttributes

An unsigned integer value which specifies the new display attributes. This may be one or more of the following values:

Constant	Description
NVT_ATTRIBUTE_NORMAL	Normal, default attributes.
NVT_ATTRIBUTE_REVERSE	Foreground and background cell colors are reversed.
NVT_ATTRIBUTE_BOLD	The character is displayed using a higher intensity color.
NVT_ATTRIBUTE_DIM	The character is displayed using a lower intensity color.
NVT_ATTRIBUTE_UNDERLINE	The character is displayed with an underline.
NVT_ATTRIBUTE_HIDDEN	The character is stored in display memory, but not shown.
NVT_ATTRIBUTE_PROTECT	The character is protected and cannot be cleared.

Return Value

If the function succeeds, the return value is the previous display attributes. If the function fails, the return value is NVT_ERROR.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[NvtGetDisplayAttributes](#), [NvtGetDisplayCell](#)

NvtSetDisplayBackColor Function

```
BOOL WINAPI NvtSetDisplayBackColor(  
    HDISPLAY hDisplay,  
    COLORREF rgbBackground  
);
```

The **NvtSetDisplayBackColor** sets the background color used by the virtual display.

Parameters

hDisplay

Handle to the virtual display.

rgbBackground

The background color specified as a 32-bit RGB value.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function sets the background color for normal, reverse, bold and reverse-bold text attributes. To set the background color for a specific attribute, use the **NvtSetDisplayColor** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[NvtGetDisplayColor](#), [NvtSetDisplayBoldColor](#), [NvtSetDisplayColor](#), [NvtSetDisplayForeColor](#)

NvtSetDisplayBoldColor Function

```
BOOL WINAPI NvtSetDisplayBoldColor(  
    HDISPLAY hDisplay,  
    COLORREF rgbBold  
);
```

The **NvtSetDisplayBoldColor** sets the color used by the virtual display to display text with the bold attribute enabled.

Parameters

hDisplay

Handle to the virtual display.

rgbBold

The bold attribute color specified as a 32-bit RGB value.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function sets the foreground color for the bold and reverse-bold text attributes. To set the color for a specific attribute, use the **NvtSetDisplayColor** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnavt10.lib

See Also

[NvtGetDisplayColor](#), [NvtSetDisplayBackColor](#), [NvtSetDisplayColor](#), [NvtSetDisplayForeColor](#)

NvtSetDisplayCell Function

```
DWORD WINAPI NvtSetDisplayCell(  
    HDISPLAY hDisplay,  
    INT xPos,  
    INT yPos,  
    DWORD dwCell  
);
```

The **NvtSetDisplayCell** function sets the value of the specified cell in the virtual display.

Parameters

hDisplay

Handle to the virtual display.

xPos

An integer value which specifies the cell column in the virtual display. A value of -1 specifies that the current cursor position should be used.

yPos

An integer value which specifies the cell row in the virtual display. A value of -1 specifies that the current cursor position should be used.

dwCell

An unsigned integer which specifies the new value for the cell in the virtual display. The low order word of this value should contain the character to be displayed at that location. The high order word should specify the attributes for that cell.

Return Value

If the function succeeds, the return value is the previous cell value. If the function fails, it returns 0xFFFFFFFF. The previous cell value is the same value returned by the **NvtGetDisplayCell** function.

Remarks

The **NvtSetDisplayCell** function is used to modify a specific character cell in the virtual display, changing both the character and the attributes for that cell. Unlike the higher level functions such as **NvtWriteDisplay** which process character strings and escape sequences, the **NvtGetDisplayCell** and **NvtSetDisplayCell** functions allow direct, low-level access to the virtual display in memory. After one or more cells are modified using this function, you should call the **NvtRefreshDisplay** function to redraw the virtual display.

The character cell attributes may be one or more of the following values:

Constant	Description
NVT_ATTRIBUTE_NORMAL	Normal, default attributes.
NVT_ATTRIBUTE_REVERSE	Foreground and background cell colors are reversed.
NVT_ATTRIBUTE_BOLD	The character is displayed using a higher intensity color.
NVT_ATTRIBUTE_DIM	The character is displayed using a lower intensity color.
NVT_ATTRIBUTE_UNDERLINE	The character is displayed with an underline.
NVT_ATTRIBUTE_HIDDEN	The character is stored in display memory, but not

	shown.
NVT_ATTRIBUTE_PROTECT	The character is protected and cannot be cleared.

One or more attributes may be combined using a bitwise Or operator. Certain attributes, such as NVT_ATTRIBUTE_BOLD and NVT_ATTRIBUTE_DIM are mutually exclusive.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnvtv10.lib

See Also

[NvtGetDisplayAttributes](#), [NvtGetDisplayCell](#), [NvtRefreshDisplay](#), [NvtSetDisplayAttributes](#), [NvtWriteDisplay](#)

NvtSetDisplayColor Function

```
BOOL WINAPI NvtSetDisplayColor(  
    HDISPLAY hDisplay,  
    UINT nColorIndex,  
    COLORREF dwColor,  
    BOOL bForeground  
);
```

The **NvtSetDisplayColor** function sets the colors used by the virtual display. Each display has a color table which specifies the foreground and background colors used when displaying text with a specific attribute.

Parameters

hDisplay

Handle to the virtual display.

nColorIndex

The index into the virtual display color table. It may be one of the following values.

Constant	Description
NVT_COLOR_NORMAL	The colors displayed for normal text. These are the default colors used with the display.
NVT_COLOR_REVERSE	The colors displayed for text with the reverse attribute set. This is only used when emulation is enabled.
NVT_COLOR_BOLD	The colors displayed for text with the bold attribute set. This is only used when emulation is enabled.
NVT_COLOR_REVERSEBOLD	The colors displayed for text with the reverse and bold attributes set. This is only used when emulation is enabled.

dwColor

The RGB color value that will be used.

bForeground

A boolean flag which specifies if the color is a foreground color, used when displaying text, or a background color.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

The default colors for the display is a black background and white foreground. Bold characters are displayed on a white background and blue foreground.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntvt10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NvtGetDisplayColor](#), [NvtSetDisplayBackColor](#), [NvtSetDisplayBoldColor](#), [NvtSetDisplayForeColor](#)

NvtSetDisplayColorMap Function

```
BOOL WINAPI NvtSetDisplayColorMap(  
    HDISPLAY hDisplay,  
    COLORREF *lpColor,  
    INT nColors  
);
```

The **NvtSetDisplayColorMap** function modifies the virtual display color table which determines what RGB values are used to display foreground and background text color attributes.

Parameters

hDisplay

Handle to the virtual display.

lpColor

A pointer to an array of COLORREF values which specifies the values to be used by the emulator when setting a color attribute. If this value is NULL, the default color table will be loaded.

nColors

The number of colors which are stored in the array. The minimum value for this parameter is 1, and the maximum value is 16.

Return Value

If the function succeeds, the return value is a non-zero value. If the function fails, the return value is zero. Failure indicates that the handle to the virtual display is invalid or the number of color values is invalid.

Remarks

When the emulator processes an escape sequence that changes the current foreground or background color, the actual RGB color value is determined by looking up the value in the virtual display's color table. The **NvtSetDisplayColorMap** function is useful for changing what values are being used when a color attribute is set. The emulator currently supports a maximum of sixteen (16) color values, and the index into the table corresponds to the color as defined by the standard for ANSI terminals:

Index	Color	Default (Hex)	Default (Integer)	Default (RGB)
0	Black	0	0	RGB(0,0,0)
1	Red	000000A0h	160	RGB(160,0,0)
2	Green	0000A000h	40960	RGB(0,160,0)
3	Yellow	0000A0A0h	41120	RGB(160,160,0)
4	Blue	00A00000h	10485760	RGB(0,0,160)
5	Magenta	00A000A0h	10485920	RGB(160,0,160)
6	Cyan	00A0A000h	10526720	RGB(0,160,160)
7	White	00E0E0E0h	14737632	RGB(224,224,224)
8	Gray	00C0C0C0h	12632256	RGB(192,192,192)

9	Light Red	008080FFh	8421631	RGB(255,128,128)
10	Light Green	0090EE90h	9498256	RGB(144,238,144)
11	Light Yellow	00C0FFFFh	12648447	RGB(255,255,192)
12	Light Blue	00E6D8ADh	15128749	RGB(173,216,230)
13	Light Magenta	00FFC0FFh	16761087	RGB(255,192,255)
14	Light Cyan	00FFFFE0h	16777184	RGB(224,255,255)
15	High White	00FFFFFFh	16777215	RGB(255,255,255)

A standard ANSI color terminal supports eight standard colors (0-7). To select a foreground color, you add 30 to the color index and pass that value as a parameter to the SGR (select graphic rendition) escape sequence. To select a background color, you add 40 to the color index. For example, to set the current foreground color to white and the background color to blue, you could send the following escape sequence:

```
ESC [ 37;44 m
```

Note that if you wanted to set the foreground color to a bold version of standard yellow, you would first set the bold attribute, and then use the index value of 3, such as:

```
ESC [ 1;33m
```

The **NvtSetDisplayColorMap** function is used to modify the actual color displayed by the emulator. For example, if the emulator processes an escape sequence which sets the current foreground color to white, the actual color displayed could be changed to light green. Passing a NULL pointer as the second parameter restores the original color map back to the default values. Note that changes to the color map will only affect new characters as they are displayed, not any previously displayed characters.

Example

The following example will load the current color table for the virtual display and change the standard white color attribute to use the same value as the high-intensity white:

```
COLORREF rgbColor[16];

if ( NvtGetDisplayColorMap(hDisplay, rgbColor, 16) )
{
    rgbColor[7] = rgbColor[15];
    NvtSetDisplayColorMap(hDisplay, rgbColor, 16);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnvtv10.lib

See Also

[NvtGetDisplayColor](#), [NvtGetDisplayColorMap](#), [NvtGetTextColor](#), [NvtSetDisplayColor](#), [NvtSetTextColor](#)

NvtSetDisplayDC Function

```
BOOL WINAPI NvtSetDisplayDC(  
    HDISPLAY hDisplay,  
    HDC hDC  
);
```

The **NvtSetDisplayDC** function sets the device context to be used by the virtual display when updating the window.

Parameters

hDisplay

Handle to the virtual display.

hDC

Handle to the device context. This parameter may be NULL, any device context that is currently associated with the virtual display will be removed. Note that the application still has the responsibility for deleting the device context, otherwise a handle leak will occur.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

It is not required that the device context be explicitly set by the application. By default, the library will use a device context created using the window attached to the virtual display. If a device context is specified by the application, it must be released when it is no longer needed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[NvtGetDisplayDC](#), [NvtUpdateDisplay](#)

NvtSetDisplayEmulation Function

```
BOOL WINAPI NvtSetDisplayEmulation(  
    HDISPLAY hDisplay,  
    UINT nEmulation  
);
```

The **NvtSetDisplayEmulation** function specifies the type of terminal emulation to be performed by the virtual display.

Parameters

hDisplay

Handle to the virtual display.

nEmulation

Identifies the virtual terminal emulation type. The following emulation types are currently supported.

Value	Description
NVT_EMULATION_NONE	The virtual display does not emulate any specific terminal type, and does not process any escape sequences.
NVT_EMULATION_ANSI	The virtual display will process ANSI escape sequences. The default keymap for an ANSI console is loaded.
NVT_EMULATION_VT100	The virtual display will process DEC VT100 escape sequences. The default keymap for a VT100 terminal is loaded.
NVT_EMULATION_VT220	The virtual display will process DEC VT220 escape sequences. The default keymap for a VT220 terminal is loaded.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

Changing the emulation type will not affect the current display.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnavt10.lib

See Also

[NvtGetDisplayEmulation](#), [NvtGetDisplayMode](#), [NvtResetDisplay](#), [NvtSetDisplayMode](#)

NvtSetDisplayFocus Function

```
BOOL WINAPI NvtSetDisplayFocus(  
    HDISPLAY hDisplay,  
    BOOL bFocus  
);
```

The **NvtSetDisplayFocus** function sets or removes the focus from the virtual display. This function should be called when the display window receives or loses focus.

Parameters

hDisplay

Handle to the virtual display.

bFocus

A boolean flag which specifies that the display should receive or lose focus.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

When the virtual display receives focus, it updates the current cursor position and displays the caret. When the display loses focus, the caret is hidden. This function should be called in response to the display window receiving the WM_SETFOCUS and WM_KILLFOCUS messages.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[NvtUpdateCaret](#)

NvtSetDisplayFont Function

```
BOOL WINAPI NvtSetDisplayFont(  
    HDISPLAY hDisplay,  
    HFONT hFont  
);
```

The **NvtSetDisplayFont** sets the font that will be used when updating the display. The specified font must be fixed-width, otherwise the virtual cursor positioning will be incorrect in some cases.

Parameters

hDisplay

Handle to the virtual display.

hFont

Handle to the font that will be used with the virtual terminal. This parameter may be NULL, in which case a default fixed-width font will be used.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

If the previous font was a default font created by the library as the result of a NULL font handle being passed to a function, it will be released. However, if the previous font was created by the application, the **DeleteObject** function must be called to release it.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[NvtGetDisplayFont](#), [NvtSetDisplayFontName](#)

NvtSetDisplayFontName Function

```
BOOL WINAPI NvtSetDisplayFontName(  
    HDISPLAY hDisplay,  
    LPTSTR lpszFontName,  
    INT nFontSize  
);
```

The **NvtSetDisplayFontName** function sets the font that is to be used when updating the display. This function always attempts to load a font which is fixed-width and uses the OEM character set.

Parameters

hDisplay

Handle to the virtual display.

lpszFontName

A pointer to a string which specifies the name of the font that will be loaded. If a NULL pointer or an empty string is passed as the value, the default **Terminal** font will be used.

nFontSize

The point size of the font that will be loaded. If this value is zero, a default point size will be used.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

The **NvtGetDisplayFont** function will return the handle to the font created by this function. However, this handle should not be released using the **DeleteObject** function. Font handles created by calling this function or passing a NULL handle to the **NvtSetDisplayFont** function will be automatically released by the library when the font is changed or the virtual display is destroyed.

If the previous display font was created using the **CreateFont** or **CreateFontIndirect** functions and then set using **NvtSetDisplayFont**, the font handle must be released by calling the **DeleteObject** function after this function has been called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnavt10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NvtGetDisplayFont](#), [NvtSetDisplayFont](#)

NvtSetDisplayForeColor Function

```
BOOL WINAPI NvtSetDisplayForeColor(  
    HDISPLAY hDisplay,  
    COLORREF rgbForeground  
);
```

The **NvtSetDisplayForeColor** sets the foreground color used by the virtual display.

Parameters

hDisplay

Handle to the virtual display.

rgbForeground

The foreground color specified as a 32-bit RGB value.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function sets the foreground color for the normal and reverse text attributes. To set the foreground color for a specific attribute, use the **NvtSetDisplayColor** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

See Also

[NvtGetDisplayColor](#), [NvtSetDisplayBackColor](#), [NvtSetDisplayBoldColor](#), [NvtSetDisplayColor](#)

NvtSetDisplayMode Function

```
BOOL WINAPI NvtSetDisplayMode(  
    HDISPLAY hDisplay,  
    UINT nMode,  
    BOOL bEnable  
);
```

The **NvtSetDisplayMode** function sets one or more display modes for the specified virtual display.

Parameters

hDisplay

Handle to the virtual display.

nMode

The virtual display mode bitmask. This value is a combination of one or more flags which determines how the emulator handles automatic line wrapping, caret display, and other functions. The following values may be specified.

Constant	Description
NVT_MODE_AUTOWRAP	The emulator will automatically wrap to the next line when a character is written to the last column on the virtual display.
NVT_MODE_SHOWCARET	The emulator will display a caret when the display receives the focus.
NVT_MODE_BLOCKCARET	The emulator will display a block caret that is the height of the selected font characters. If this mode is not set, the caret is displayed as an underline.
NVT_MODE_BELL	The emulator will beep when a BEL character is processed.
NVT_MODE_CRLF	The cursor will automatically be positioned at the first column when a linefeed character is processed.
NVT_MODE_CRNL	The cursor will automatically advance to the next row when a carriage return character is processed.
NVT_MODE_APPCURSOR	The emulator cursor keys are placed in application mode. This mode changes the default key mappings used when the cursor (arrow) keys are translated. This corresponds to the application mode supported by DEC VT terminals.
NVT_MODE_APPKEYPAD	The emulator keypad keys are placed in application mode. This mode changes the default key mappings used when the keypad keys are translated. This corresponds to the application keypad mode supported by DEC VT terminals.
NVT_MODE_ORIGIN	The emulator is in origin mode. If enabled, the cursor cannot be positioned outside of the current scrolling region. Otherwise, the cursor can be positioned at any valid location on the virtual display.
NVT_MODE_COLOR	The emulator supports the use of escape sequences to

	change the foreground and background colors. This option is enabled by default if emulating an ANSI console or DEC VT220 terminal.
NVT_MODE_TABOVER	The emulator will clear the character cells between the current cursor position and the next tab stop when the HT (horizontal tab) control sequence is processed. By default this mode is disabled, and the cursor is simply positioned at the next tab stop.
NVT_MODE_HSCROLL	The emulator will display a horizontal scroll bar if the number of visible columns are less than the total number of columns in the virtual display. Disabling this mode prevents a horizontal scroll bar from being displayed, regardless of the number of visible columns. By default, this mode is enabled.
NVT_MODE_VSCROLL	The emulator will display a vertical scroll bar if the number of visible rows are less than the total number of rows in the virtual display. Disabling this mode prevents a vertical scroll bar from being displayed, regardless of the number of visible rows. By default, this mode is enabled.
NVT_MODE_NOREFRESH	The emulator will not automatically refresh the window after any change has been made to the virtual display, including changes in the cursor position or display mode. This allows the caller to make a sequence of changes, and then update the display all at one time to prevent a flicker effect.

bEnable

This boolean flag specifies if the specified mode is to be enabled or disabled.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnavt10.lib

See Also

[NvtCreateDisplay](#), [NvtGetDisplayInfo](#), [NvtGetDisplayMode](#), [NvtSetDisplayEmulation](#)

NvtSetDisplayScrollPos Function

```
BOOL WINAPI NvtSetDisplayScrollPos(  
    HDISPLAY hDisplay,  
    INT nScrollBar,  
    INT nScrollPos  
);
```

The **NvtSetDisplayScrollPos** function sets the position of the scroll box for the specified scroll bar and redraws the scroll bar to reflect the new position of the scroll box.

Parameters

hDisplay

Handle to the virtual display.

nScrollBar

Specifies the scroll bar to be set. This parameter can be one of the following values:

Value	Description
SB_HORZ	Sets the position of the scroll box in the display's standard horizontal scroll bar.
SB_VERT	Sets the position of the scroll box in the display's standard vertical scroll bar.

nScrollPos

Specifies the new row or column of the scroll box. The position must be within the scrolling range.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function should always be used to change the scroll box position. The **NvtSetDisplayScrollPos** function will result in unpredictable behavior if used on the virtual display window.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csnavt10.lib

See Also

[NvtGetDisplayScrollPos](#), [NvtScrollDisplay](#), [NvtUpdateDisplay](#)

NvtSetDisplaySize Function

```
BOOL WINAPI NvtSetDisplaySize(  
    HDISPLAY hDisplay,  
    LPSIZE lpSize  
);
```

The **NvtSetDisplaySize** sets the size of the virtual display.

Parameters

hDisplay

Handle to the virtual display.

lpRect

Pointer to a SIZE structure which specifies the new size of the virtual display. The width should be specified in columns and the height should be specified in rows.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[NvtGetDisplayInfo](#), [NvtGetDisplayRect](#)

NvtSetDisplayWindow Function

```
BOOL WINAPI NvtSetDisplayWindow(  
    HDISPLAY hDisplay,  
    HWND hWnd  
);
```

The **NvtSetDisplayWindow** function sets the window used by the virtual display.

Parameters

hDisplay

Handle to the virtual display.

hWnd

Handle to the window that will be used by the virtual display. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function will clear the display and reset the current text attributes.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[NvtCreateDisplay](#), [NvtGetDisplayDC](#), [NvtGetDisplayWindow](#), [NvtResetDisplay](#), [NvtSetDisplayDC](#)

NvtSetMappedKey Function

```
BOOL WINAPI NvtSetMappedKey(  
    HDISPLAY hDisplay,  
    UINT nMappedKey,  
    LPCTSTR LpszKeyBuffer  
);
```

The **NvtSetMappedKey** function maps an escape sequence to the specified key.

Parameters

hDisplay

Handle to the virtual display.

nMappedKey

The key which is an index into the display key mapping table. This defines the values returned by special (function) keys for the current emulation.

lpszKeyBuffer

Pointer to a string which defines the sequence of characters that are to be mapped to the specified key. Passing an empty string or NULL pointer will delete the current sequence mapped to the key and restore the default value, if one has been defined.

Return Value

If the function succeeds, the return value is non-zero and the escape sequence is mapped to the specified key. If the key value is invalid or could not be mapped, then the function will return zero.

Remarks

The **NvtSetMappedKey** function maps an escape sequence to a special key. This function can be used to specify what sequence of characters should be sent in response to a keypress (for example, what sequence should be sent to a server when the user presses the F1 function key). There are a number of default sequences that are mapped to the function and cursor keys, based on the current emulation. Calling this function will override the default sequence for a key, if one has been defined.

The following special keys are defined:

Value	Constant	Description	Value	Constant	Description
0	NVT_F1	F1 function key	26	NVT_UP	Cursor up key
1	NVT_F2	F2 function key	27	NVT_DOWN	Cursor down key
2	NVT_F3	F3 function key	28	NVT_LEFT	Cursor left key
3	NVT_F4	F4 function key	29	NVT_RIGHT	Cursor right key
4	NVT_F5	F5 function key	30	NVT_INSERT	Insert key
5	NVT_F6	F6 function key	31	NVT_DELETE	Delete key
6	NVT_F7	F7 function key	32	NVT_HOME	Home key
7	NVT_F8	F8 function key	33	NVT_END	End key
8	NVT_F9	F9 function key	34	NVT_PGUP	Page up key
9	NVT_F10	F10 function key	35	NVT_PGDN	Page down key
10	NVT_F11	F11 function key	36	NVT_APPUP	Up arrow key
11	NVT_F12	F12 function key	37	NVT_APPDOWN	Down arrow key
12	NVT_SF1	Shift F1 function key	38	NVT_APPLEFT	Left arrow key
13	NVT_SF2	Shift F2 function key	39	NVT_APPRIGHT	Right arrow key

14	NVT_SF3	Shift F3 function key	40	NVT_APPENTER	Keypad enter key
15	NVT_SF4	Shift F4 function key	41	NVT_KEYPAD0	Numeric keypad 0
16	NVT_SF5	Shift F5 function key	42	NVT_KEYPAD1	Numeric keypad 1
17	NVT_SF6	Shift F6 function key	43	NVT_KEYPAD2	Numeric keypad 2
18	NVT_SF7	Shift F7 function key	44	NVT_KEYPAD3	Numeric keypad 3
19	NVT_SF8	Shift F8 function key	45	NVT_KEYPAD4	Numeric keypad 4
20	NVT_SF9	Shift F9 function key	46	NVT_KEYPAD5	Numeric keypad 5
21	NVT_SF10	Shift F10 function key	47	NVT_KEYPAD6	Numeric keypad 6
22	NVT_SF11	Shift F11 function key	48	NVT_KEYPAD7	Numeric keypad 7
23	NVT_SF12	Shift F12 function key	49	NVT_KEYPAD8	Numeric keypad 8
24	NVT_ENTER	Enter key	50	NVT_KEYPAD9	Numeric keypad 9
25	NVT_ERASE	Backspace key			

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csntv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NvtGetDisplayMode](#), [NvtGetMappedKey](#), [NvtResetMappedKeys](#), [NvtSetDisplayMode](#), [NvtTranslateMappedKey](#)

NvtSetScrollRegion Function

```
BOOL WINAPI NvtSetScrollRegion(  
    HDISPLAY hDisplay,  
    INT nTop,  
    INT nBottom  
);
```

The **NvtSetScrollRegion** function sets the top and bottom rows of the current scrolling region.

Parameters

hDisplay

Handle to the virtual display.

nTop

The top row of the current scrolling region. If this value is greater than the bottom scrolling region row or the total number of rows in the display, it will be silently adjusted.

nBottom

The bottom row of the current scrolling region. If this value is less than zero or the top scrolling region row, it will be silently adjusted.

Return Value

If the function succeeds, it will return zero. If the handle to the display is invalid, the function will return zero.

Remarks

The **NvtSetScrollRegion** function allows an application to set the current scrolling region for the virtual display. This specifies the region (between the top and bottom rows) in which text will normally scroll. If the display is in origin mode, the cursor cannot be positioned outside of the scrolling region.

The minimum scrolling region that may be defined is two rows. If a scrolling region is specified that is less than two rows, the function will fail and the current scrolling region will remain unchanged. Specifying values of -1 for both arguments will reset the scrolling region to the default values (the full display).

The DEC STBM escape sequence is used to set or clear the scrolling region of the virtual display.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnavt10.lib

See Also

[NvtGetDisplayMode](#), [NvtSetDisplayMode](#), [NvtSetScrollRegion](#)

NvtSetTextColor Function

```
BOOL WINAPI NvtSetTextColor(  
    HDISPLAY hDisplay,  
    COLORREF rgbColor,  
    BOOL bForeground  
);
```

The **NvtSetTextColor** function changes the current foreground or background text color.

Parameters

hDisplay

Handle to the virtual display.

rgbColor

A color value which specifies the current foreground or background text color. The RGB macro can be used to specify the red, green and blue values for the color.

bForeground

A boolean value which determines if the foreground or background color is changed. A value of TRUE indicates that the foreground color should be changed, otherwise the background color is changed.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. Failure indicates that the handle to the display is invalid or that the current display does not support color text attributes.

Remarks

This function is used to change the current foreground or background color, as determined by the text attribute. Note that changing the current foreground or background text color does not affect the virtual display color table. To change how color attributes are mapped to an RGB color value, use the **NvtSetDisplayColorMap** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[NvtGetDisplayColorMap](#), [NvtGetTextColor](#), [NvtSetDisplayColorMap](#)

NvtTranslateMappedKey Function

```
BOOL WINAPI NvtTranslateMappedKey(  
    HDISPLAY hDisplay,  
    UINT nKey,  
    UINT nFlags,  
    UINT * lpnMappedKey,  
    LPTSTR lpszKeyBuffer,  
    UINT cchBuffer  
);
```

The **NvtTranslateMappedKey** function translates a virtual key press to an escape sequence based on the current terminal emulation that has been selected.

Parameters

hDisplay

Handle to the virtual display.

nKey

The virtual key code for the specified key.

nFlags

The scan code, key-transition code, previous key state, and context code for the specified key.

lpnMappedKey

A pointer to an unsigned integer which will contain the index into the keymap table when the function returns. This is the same value used with the **NvtGetMappedKey** and **NvtSetMappedKey** function. If the index into the keymap is not required, this parameter can be NULL.

lpszKeyBuffer

Address of the buffer to receive the escape sequence mapped to the specified key. If the mapped key string is not required, this parameter can be NULL.

cchBuffer

The maximum number of characters that can be copied into the key buffer, including the terminating null character. If the *lpszKeyBuffer* parameter is NULL, this value must be zero.

Return Value

If the virtual key can be mapped to an escape sequence, the function will return a non-zero value. If the key is not mapped, or one of the arguments is invalid, the function will return zero.

Remarks

The **NvtTranslateMappedKey** function allows an application to map a virtual key code to an escape sequence that is appropriate for the type of terminal that is being emulated. For example, it will return the escape sequence for the F1 function key when passed the VK_F1 key value. This function should be called when the WM_KEYDOWN message is processed by an application so that it may send the correct sequence to the server.

This function should only be called in response to a keyboard message such as WM_KEYDOWN. To determine if a specific key has been mapped to an escape sequence, use the **NvtGetMappedKey** function.

Example

```

case WM_KEYDOWN:
    /*
     * If the Num Lock key is pressed, then set the terminal into application
     * keypad mode. This will change how the NvtTranslateMappedKey function will
     * translate the keypad keys.
     *
     * Note that the terminal may also be placed into application keypad mode
     * if emulating a DEC VT terminal and the DECNMK escape sequence is sent
     * by the host.
     */
    if (wParam == VK_NUMLOCK)
        NvtSetDisplayMode(hDisplay, NVT_MODE_APPKEYPAD, !
(GetKeyState(VK_NUMLOCK) & 1));
    else
    {
        BOOL bMapped;
        UINT nMappedKey;
        TCHAR szKey[128];

        bMapped = NvtTranslateMappedKey(hDisplay, wParam, HIWORD(lParam),
&nMappedKey, szKey, 128);

        if (bMapped)
            TelnetWrite(hClient, szKey, lstrlen(szKey));
    }
    break;

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnavt10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[NvtGetDisplayMode](#), [NvtGetMappedKey](#), [NvtResetMappedKeys](#), [NvtSetDisplayMode](#),
[NvtSetMappedKey](#)

NvtUninitialize Function

```
VOID WINAPI NvtUninitialize();
```

The **NvtUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **NvtInitialize** call before it can call any of the other library functions. When it has completed the use of library, the application must call **NvtUninitialize** to allow the library to free any resources allocated on behalf of the process.

There must be a call to **NvtUninitialize** for every successful call to **NvtInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnavt10.lib

See Also

[NvtDestroyDisplay](#), [NvtUninitialize](#)

NvtUpdateCaret Function

```
BOOL WINAPI NvtUpdateCaret(  
    HDISPLAY hDisplay  
);
```

The **NvtUpdateCaret** function updates the position of the caret in the display window to the current cursor position.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: csnavt10.lib

See Also

[NvtSetCursorPos](#), [NvtSetDisplayMode](#), [NvtUpdateDisplay](#)

NvtUpdateDisplay Function

```
BOOL WINAPI NvtUpdateDisplay(  
    HDISPLAY hDisplay  
);
```

The **NvtUpdateDisplay** function updates the window attached to the virtual display.

Parameters

hDisplay

Handle to the virtual display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function should be called when the display window receives a WM_PAINT message. If the **NvtSetDisplayDC** function has not been called to explicitly set the display device context, this function will acquire one for the display window. In this case, the application should not create a device context for the window or call the **BeginPaint** and **EndPaint** functions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: csntv10.lib

See Also

[NvtRefreshDisplay](#), [NvtResizeDisplay](#), [NvtSetDisplayDC](#), [NvtSetDisplayScrollPos](#),
[NvtSetDisplayWindow](#)

NvtWriteDisplay Function

```
BOOL WINAPI NvtWriteDisplay(  
    HDISPLAY hDisplay,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **NvtWriteDisplay** function writes the contents of the specified buffer to the virtual display.

Parameters

hDisplay

Handle to the virtual display.

lpBuffer

Pointer to the buffer which contains the data to be written to the virtual display.

cbBuffer

Number of bytes to write to the display.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, it returns zero.

Remarks

This function writes the data at the current cursor location. Control characters are recognized by this function and processed accordingly. If ANSI emulation is enabled, embedded escape sequences will also be parsed and processed.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: csntv10.lib

See Also

[NvtGetDisplayText](#), [NvtRefreshDisplay](#), [NvtUpdateCaret](#), [NvtUpdateDisplay](#)

Terminal Emulation Data Structures

- INITDATA
- NVTDISPLAYINFO

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

INITDATA Structure

This structure contains information about the SocketTools library when the initialization function returns.

```
typedef struct _INITDATA
{
    DWORD        dwSize;
    DWORD        dwVersionMajor;
    DWORD        dwVersionMinor;
    DWORD        dwVersionBuild;
    DWORD        dwOptions;
    DWORD_PTR    dwReserved1;
    DWORD_PTR    dwReserved2;
    TCHAR        szDescription[128];
} INITDATA, *LPINITDATA;
```

Members

dwSize

Size of this structure. This structure member must be set to the size of the structure prior to calling the initialization function. Failure to do so will cause the function to fail.

dwVersionMajor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the major version number for the library.

dwVersionMinor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the minor version number for the library.

dwVersionBuild

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the build number for the library.

dwOptions

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved1

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved2

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

szDescription

A null terminated string which describes the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

NVTDISPLAYINFO Structure

This structure is used by the [NvtGetDisplayInfo](#) function to return information about the specified virtual terminal display. No member of this structure should be modified directly by the application.

```
typedef struct _NVTDISPLAYINFO {
    HWND    hWnd;
    HFONT   hFont;
    INT     xPos;
    INT     yPos;
    INT     cxClient;
    INT     cyClient;
    INT     cxChar;
    INT     cyChar;
    INT     nScrollCol;
    INT     nScrollRow;
    INT     nMaxScrollCol;
    INT     nMaxScrollRow;
} NVTDISPLAYINFO, *LPNVTDISPLAYINFO;
```

Members

hWnd

The handle to the terminal emulation display window.

hFont

The handle to the current font.

xPos

The current display x coordinate.

yPos

The current display y coordinate.

cxClient

The width of the client window in pixels.

cyClient

The height of the client window in pixels.

cxChar

The width of the current font in pixels.

cyChar

The height of the current font in pixels.

nScrollCol

The current horizontal scrolling column.

nScrollRow

The current vertical scrolling row.

nMaxScrollCol

The maximum horizontal scrolling column.

nMaxScrollRow

The maximum vertical scrolling row.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Terminal Emulator Control Sequences

Terminal Control Sequences

<ESC>c	Reset display to initial state
<ESC>8	Display alignment test

Cursor Control Sequences

<ESC>D	Move cursor down to next line
<ESC>E	Move cursor to first column and down one line
<ESC>M	Move cursor up one line
<ESC>7	Save cursor position, attributes and colors
<ESC>8	Restore saved cursor position, attributes and colors
<ESC>[nA	Move cursor up <i>n</i> lines
<ESC>[nB	Move cursor down <i>n</i> lines
<ESC>[nC	Move cursor forward <i>n</i> spaces
<ESC>[nD	Move cursor backward <i>n</i> spaces
<ESC>[nE	Move cursor to beginning of line, down <i>n</i> lines
<ESC>[nF	Move cursor to beginning of line, up <i>n</i> lines
<ESC>[xG	Move cursor to column <i>x</i>
<ESC>[y;xH	Move cursor to line <i>y</i> , column <i>x</i>
<ESC>[nI	Move cursor forward <i>n</i> tabstops
<ESC>[nZ	Move cursor backwards <i>n</i> tabstops
<ESC>[na	Move cursor forward <i>n</i> spaces
<ESC>[yd	Move cursor to row <i>y</i>
<ESC>[ne	Move cursor down <i>n</i> lines
<ESC>[y;xf	Move cursor to line <i>y</i> , column <i>x</i>
<ESC>[s	Save cursor position
<ESC>[u	Return to saved cursor position
<ESC>[x`	Move cursor to column <i>x</i>

Attribute and Color Sequence

Select display attributes and color

<i>n</i> Value	Description
0	Reset to default attributes and colors
1	Bold attribute
2	Dim attribute
4	Underline attribute
5	Blink attribute (same as reverse)
7	Reverse attribute
8	Hidden attribute
22	Clear bold attribute
24	Clear underline attribute
25	Clear blink attribute

	27	Clear reverse attribute
	29	Clear color attributes
<ESC>[<i>nm</i>	30	Black foreground
	31	Red foreground
	32	Green foreground
	33	Yellow foreground
	34	Blue foreground
	35	Magenta foreground
	36	Cyan foreground
	37	White foreground
	40	Black background
	41	Red background
	42	Green background
	43	Yellow background
	44	Blue background
	45	Magenta background
	46	Cyan background
	47	White background

Character Set Sequences

<ESC>(A	Assign ISO Latin 1 character set to font bank G0
<ESC>(B	Assign United States ASCII character set to font bank G0
<ESC>(0	Assign graphics character set to font bank G0
<ESC>)A	Assign ISO Latin 1 character set to font bank G1
<ESC>)B	Assign United States ASCII character set to font bank G1
<ESC>)0	Assign graphics character set to font bank G1

Erase Sequences

<ESC>[*n*@ Insert *n* blank spaces
 Erase all or part of the display

<i>n</i> Value	Description
----------------	-------------

<ESC>[<i>n</i> J	0	From current position to end of display
	1	From beginning of display to current position
	2	Erase the entire display

Erase all or part of a line

<i>n</i> Value	Description
----------------	-------------

<ESC>[<i>n</i> K	0	From current position to end of line
	1	From beginning of line to current position
	2	Erase the entire line

<ESC>[*n*L Insert *n* new blank lines

<ESC>[nM Delete *n* lines from current cursor position
<ESC>[nP Delete *n* characters from current cursor position

Scrolling Sequences

<ESC>[nS Scroll display up *n* lines
<ESC>[nT Scroll display down *n* lines
<ESC>[nX Erase *n* characters from the current position
<ESC>[y1;y2r Set scrolling region from lines *y1* to *y2*

Keypad Sequences

<ESC>= Place keypad into applications mode
<ESC>> Place keypad into numeric mode

Emulation Option Sequences

Set emulation option

<i>n</i> Value	Description
----------------	-------------

<ESC>[?nh	1	Enable cursor key application mode
	2	Enable ANSI escape sequences
	5	Reverse foreground and background colors
	6	Enable origin mode
	7	Enable auto-wrap mode
	20	Enable linefeed/newline mode
	25	Display caret
	66	Place keypad in applications mode

Set emulation option

<i>n</i> Value	Description
----------------	-------------

<ESC>[?n1	1	Disable cursor key application mode
	2	Enable VT52 escape sequences
	5	Restore foreground and background colors
	6	Disable origin mode
	7	Disable auto-wrap mode
	20	Disable linefeed/newline mode
	25	Hide caret
	66	Place keypad in numeric mode

Console Escape Sequences

<ESC>[=nA Set the overscan color (ignored)
<ESC>[=n1;n2B Set bell sound (parameters ignored)
<ESC>[=n1;n2C Set the caret size
Set background color intensity

<ESC>[=nD	<i>n</i> Value	Description
	0	Decrease background color intensity

	1	Increase background color intensity
<ESC>[= <i>n</i> E		Set blink vs. bold attribute (ignored)
<ESC>[= <i>n</i> F		Set normal foreground color
<ESC>[= <i>n</i> G		Set normal background color
<ESC>[= <i>n</i> H		Set reverse foreground color
<ESC>[= <i>n</i> I		Set reverse background color
<ESC>[= <i>n</i> J		Set graphics foreground color
		Set graphics background color

	<i>n</i>	Value	Description
	0		Black
	1		Blue
	2		Green
	3		Cyan
	4		Red
	5		Magenta
<ESC>[= <i>n</i> K	6		Brown
	7		White
	8		Gray
	9		Light blue
	10		Light green
	11		Light cyan
	12		Light red
	13		Light magenta
	14		Yellow
	15		High White

Control Character Sequences

<CTL>G	Ring audible bell, if enabled
<CTL>H	Move cursor one character backwards
<CTL>I	Move cursor forward to next tabstop
<CTL>J	Move cursor down to next line
<CTL>M	Move cursor to beginning of line
<CTL>N	Select G1 character set
<CTL>O	Select G0 character set
<CTL>Z	Abort current escape sequence
	Erase and move cursor one character backwards

Text Message Library

Send text messages to a mobile communications device using a gateway service.

Reference

- [Functions](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

File Name	CSTXTV10.DLL
Version	10.0.1468.2518
LibID	224FEFCC-A572-4432-81E0-FC25C4ED00AA
Import Library	CSTXTV10.LIB
Dependencies	None

Overview

Short Message Service (SMS) is a text messaging service used by mobile communication devices to exchange brief text messages. Most service providers also provide gateway servers that can be used to send messages to a wireless device on their network using standard email protocols. The Text Message API provides functions that can be used to determine the provider associated with a specific telephone number and send a text message to the device using the provider's mail gateway.

This library has been designed to assist developers in sending text message notifications as part of their application. For example, it can be used to enable your software to automatically send notifications when a specific event occurs, such as an error condition. This library is not designed to be used with software that will send out a large number of text messages to many users, and there are limitations on the number of messages that may be sent to different phone numbers over a short period of time. Because many recipients must pay a fee for each text message they receive, text messages should only be sent to those who explicitly request them.

Note: This library only supports service providers in North America and cannot be used to send text messages to mobile devices that use providers outside of the United States and Canada. Some service providers may prevent messages from being sent through their gateway to a user that does not have unlimited text messaging as part of their service agreement.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Text Message Functions

Function	Description
SmsDisableTrace	Disable logging of network function calls
SmsEnableTrace	Enable logging of network function calls to a text file
SmsEnumProviders	Enumerate the available wireless service providers
SmsGetErrorString	Return a description for the specified error code
SmsGetFirstProvider	Return information about the first supported wireless service provider
SmsGetGateway	Return information about the gateway server for the specified phone number
SmsGetNextProvider	Return information about the next supported wireless service provider
SmsGetProvider	Return information about the wireless service provider for the specified phone number
SmsGetLastError	Return the last error code
SmsInitialize	Initialize the library and validate the specified license key at runtime
SmsSendMessage	Send a text message to the specified mobile device
SmsSetLastError	Set the last error code
SmsUninitialize	Terminate use of the library by the application

SmsDisableTrace Function

```
BOOL WINAPI SmsDisableTrace();
```

The **SmsDisableTrace** function disables the logging of network function calls.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstxtv10.lib

See Also

[SmsEnableTrace](#)

SmsEnableTrace Function

```
BOOL WINAPI SmsEnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **SmsEnableTrace** function enables the logging of network function calls to a text file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace function logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmsDisableTrace](#)

SmsEnumProviders Function

```
INT WINAPI SmsEnumProviders(  
    LPSMSPROVIDER lpProviders,  
    INT nMaxProviders,  
    DWORD dwReserved  
);
```

The **SmsEnumProviders** function enumerates the supported wireless service providers and populates an array of **SMSPROVIDER** structures.

Parameters

lpProviders

A pointer to an array of **SMSPROVIDER** structures that will be populated with information about each service provider. If this parameter is NULL, the function will return the number of service providers that are known.

nMaxProviders

An integer value which specifies the maximum number of service providers that may be enumerated by this function. If this parameter is zero, the function will return the number of service providers that are known. If the *lpProviders* parameter is NULL, this value should be zero.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is the number of known service providers. If the function fails, the return value is **SMS_ERROR**. To get extended error information, call **SmsGetLastError**. If the *lpProviders* parameter is not NULL and the *nMaxProviders* parameter indicates the array is not large enough to store all of the provider information, this function will fail with an error indicating that the buffer is too small.

Remarks

The **SmsEnumProviders** function is used to enumerate all of the supported service providers, populating an array of **SMSPROVIDER** structures that contains information about each provider, such as their name, domain, region of the country they service and the maximum message size they will accept. Typically this would be used to update a user interface control such as a listbox or drop-down combobox, enabling a user to select a preferred service provider.

Because some programming languages may not support arrays of structures in the same way that C/C++ does, the **SmsGetFirstProvider** and **SmsGetNextProvider** functions offer an alternative way to easily enumerate the available service providers.

To obtain information about a single service provider, use the **SmsGetProvider** function.

Example

```
SMSPROVIDER smsProviders[MAXPROVIDERS];  
INT nProviders;  
  
nProviders = SmsEnumProviders(smsProviders, MAXPROVIDERS, 0);  
if (nProviders == SMS_ERROR)  
{  
    DWORD dwError = SmsGetLastError();
```

```
    _tprintf(_T("An error has occurred, error code %d\n"),
(INT)LOWORD(dwError));
}
else
{
    for (INT nIndex = 0; nIndex < nProviders; nIndex++)
        _tprintf(_T("%s\n"), smsProviders[nIndex].szName);

    _tprintf(_T("%d providers returned\n"), nProviders);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmsGetFirstProvider](#), [SmsGetGateway](#), [SmsGetNextProvider](#), [SmsGetProvider](#), [SMSPROVIDER](#)

SmsGetErrorString Function

```
INT WINAPI SmsGetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);
```

The **SmsGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **SmsGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The last-error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstxtv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmsGetLastError](#), [SmsSetLastError](#)

SmsGetFirstProvider Function

```
BOOL WINAPI SmsGetFirstProvider(  
    LPSMSPROVIDER lpProvider,  
    LPDWORD lpdwToken,  
);
```

The **SmsGetFirstProvider** function returns information about the first supported wireless service provider.

Parameters

lpProvider

A pointer to an [SMSPROVIDER](#) structure that will contain information about the service provider. This parameter cannot be NULL.

lpdwToken

A pointer to an unsigned integer value that is used when enumerating the service providers. The value will be initialized by this function and the modified by subsequent calls to the **SmsGetNextProvider** function. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **SmsGetLastError**.

Remarks

The **SmsGetFirstProvider** function is used in conjunction with **SmsGetNextProvider** to enumerate all of the supported wireless providers available to the client. These two functions can be used as an alternative to the **SmsEnumProviders** function, which may not be easily used with some programming languages because it populates an array of data structures rather than a single structure. The data that is returned is identical, the only difference is the method by which the service providers are enumerated.

The *lpdwToken* parameter is used to maintain context between multiple calls to the **SmsGetNextProvider** function. It should be treated as an opaque value and never modified directly by the application.

Example

```
SMSPROVIDER smsProvider;  
DWORD dwToken;  
BOOL bResult;  
  
bResult = SmsGetFirstProvider(&smsProvider, &dwToken);  
while (bResult)  
{  
    pListBox->AddString(smsProvider.szName);  
    bResult = SmsGetNextProvider(&smsProvider, &dwToken);  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmsEnumProviders](#), [SmsGetGateway](#), [SmsGetNextProvider](#), [SMSPROVIDER](#)

SmsGetGateway Function

```
INT WINAPI SmsGetGateway(  
    LPCTSTR lpszPhoneNumber,  
    LPCTSTR lpszProvider,  
    LPSMSGATEWAY lpGateway  
);
```

The **SmsGetGateway** function returns text message service information for a phone number.

Parameters

lpszPhoneNumber

A pointer to a string which specifies the telephone number that you wish to obtain information about. Any whitespace, punctuation or other non-numeric characters in the string will be ignored. This parameter cannot be NULL.

lpszProvider

A pointer to a string which specifies the preferred service provider for this telephone number. If the preferred service provider is unknown, this parameter can be NULL and the default provider will be selected.

lpGateway

A pointer to an [SMSGATEWAY](#) structure that will contain information about the text message gateway when the function returns. This includes information such as the name of the provider, the server that will accept text messages for this phone number, and the recipient address that should be used. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMS_ERROR. To get extended error information, call **SmsGetLastError**.

Remarks

The **SmsGetGateway** function returns information about the service provider and mail gateway for a specific phone number, and can be used to determine if a given phone number is assigned to a mobile device capable of receiving text messages. This is done by sending an query to a server that will check the phone number against a database of known providers and the phone numbers that have been allocated for wireless devices. If the phone number is valid, information will be returned about the provider that is responsible for that number along with information about its text message gateway service.

If the *lpszProvider* parameter is not NULL, this will identify a preferred provider for the phone number specified. In the United States and Canada, most wireless common carriers are required to provide wireless number portability (WNP) which allows a customer to continue to use their current phone number even if they switch to another service provider. This can result in a situation where a specific phone number is shown as allocated to one provider, but in actuality that user has switched to a different provider. For example, a user may have originally purchased a phone and service with AT&T and then later switched to Verizon, but decided to keep their phone number. In this case, if Verizon was not specified as the preferred provider, the library would attempt to send the message to the AT&T gateway, since that was the original provider who allocated the phone number.

For most applications, the correct way to handle the situation in which a user may have switched to a different service provider is to allow them to select an alternate service provider in your user

interface. For example, you could display a drop-down list of available service providers, populated using the **SmsEnumProviders** function. If they select a preferred provider, then you would pass that value to this function. If they do not, then specify a NULL pointer and the default provider will be selected.

This function sends an HTTP query to the server **api.sockettools.com** to obtain information about the phone number and wireless service provider. This requires that the local system can establish a standard network connection over port 80. If the client cannot connect to the server, the function will fail and an appropriate error will be returned. The server imposes a limit on the maximum number of connections that can be established and the maximum number of requests that can be issued per minute. If this function is called multiple times over a short period, the library may also force the application to block briefly. Server responses are cached per session, so calling this function multiple times using the same phone number will not increase the request count.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmsEnumProviders](#), [SmsGetProvider](#), [SMSGATEWAY](#)

SmsGetLastError Function

```
DWORD WINAPI SmsGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SmsSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **SmsGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **SmsSetLastError(0)** when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CHANNEL or SMS_ERROR. Those functions which call **SmsSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

See Also

[SmsGetErrorString](#), [SmsSetLastError](#)

SmsGetNextProvider Function

```
BOOL WINAPI SmsGetNextProvider(  
    LPSMSPROVIDER lpProvider,  
    LPDWORD lpdwToken,  
);
```

The **SmsGetNextProvider** function returns information about the next supported wireless service provider.

Parameters

lpProvider

A pointer to an [SMSPROVIDER](#) structure that will contain information about the service provider. This parameter cannot be NULL.

lpdwToken

A pointer to an unsigned integer value that is used when enumerating the service providers. The value will be initialized by the **SmsGetFirstProvider** function and the modified by subsequent calls to this function. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the last service provider has been enumerated or the function fails, the return value is zero. To get extended error information, call **SmsGetLastError**.

Remarks

The **SmsGetNextProvider** function is used in conjunction with **SmsGetFirstProvider** to enumerate all of the supported wireless providers available to the client. These two functions can be used as an alternative to the **SmsEnumProviders** function, which may not be easily used with some programming languages because it populates an array of data structures rather than a single structure. The data that is returned is identical, the only difference is the method by which the service providers are enumerated.

The *lpdwToken* parameter is used to maintain context between multiple calls to this function. It should be treated as an opaque value and never modified directly by the application.

Example

```
SMSPROVIDER smsProvider;  
DWORD dwToken;  
BOOL bResult;  
  
bResult = SmsGetFirstProvider(&smsProvider, &dwToken);  
while (bResult)  
{  
    pListBox->AddString(smsProvider.szName);  
    bResult = SmsGetNextProvider(&smsProvider, &dwToken);  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmsEnumProviders](#), [SmsGetGateway](#), [SmsGetNextProvider](#), [SMSPROVIDER](#)

SmsGetProvider Function

```
INT WINAPI SmsGetProvider(  
    LPCTSTR lpszPhoneNumber,  
    LPSMSPROVIDER lpProvider  
);
```

The **SmsGetProvider** function returns information about the service provider for the specified phone number.

Parameters

lpszPhoneNumber

A pointer to a string which specifies the telephone number that you wish to obtain information about. Any whitespace, punctuation or other non-numeric characters in the string will be ignored. This parameter cannot be NULL.

lpProvider

A pointer to an [SMSPROVIDER](#) structure that will contain information about the service provider for the specified phone number.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMS_ERROR. To get extended error information, call **SmsGetLastError**.

Remarks

The **SmsGetProvider** function returns information about the service provider associated with a phone number. This is done by sending an query to a server that will check the phone number against a database of known providers and the phone numbers that have been allocated for wireless devices. If the phone number is valid, information will be returned about the provider that is responsible for that number.

Because most wireless carriers in the United States and Canada must provide for wireless number portability, there is the possibility that the provider information returned may no longer correspond to the telephone number. It is recommended that you provide your end-user with the ability to specify an alternate preferred provider to use when sending the text message. For more information, refer to the **SmsGetGateway** function.

This function sends an HTTP query to the server **api.sockettools.com** to obtain information about the wireless service provider. This requires that the local system can establish a standard network connection over port 80. If the client cannot connect to the server, the function will fail and an appropriate error will be returned. The server imposes a limit on the maximum number of connections that can be established and the maximum number of requests that can be issued per minute. If this function is called multiple times over a short period, the library may also force the application to block briefly. Server responses are cached per session, so calling this function multiple times using the same phone number will not increase the request count.

For a list of all supported wireless service providers, use the **SmsEnumProviders** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmsEnumProviders](#), [SmsGetGateway](#), [SMS PROVIDER](#)

SmsInitialize Function

```
BOOL WINAPI SmsInitialize(  
    LPCTSTR lpszLicenseKey,  
    LPINITDATA lpData  
);
```

The **SmsInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpData

Pointer to an INITDATA data structure. This parameter may be NULL if the initialization data for the library is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **SmsGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must insure that this function is called with a valid license key.

If the *lpData* argument is specified, it must point to an INITDATA structure which has been initialized by setting the *dwSize* member to the size of the structure. All other structure members should be set to zero. If the function is successful, then the INITDATA structure will be filled with identifying information about the library.

Although it is only required that **SmsInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **SmsUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **SmsInitialize** or **SmsUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

SmsSendMessage Function

```
INT WINAPI SmsSendMessage(  
    LPSSERVICE lpService,  
    LPSMSMESSAGE lpMessage,  
    DWORD dwReserved,  
);
```

The **SmsSendMessage** function sends a text message to the specified mobile device.

Parameters

lpService

A pointer to an [SMSERVICE](#) structure that identifies the messaging service that will be used to send the text message. The default service sends the message through the mail server gateway for the wireless service provider associated with the recipient's phone number. This parameter cannot be NULL.

lpMessage

A pointer to an [SMSMESSAGE](#) structure that contains information about the message to be sent, including the sender, the recipient and the text message itself. This parameter cannot be NULL.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is SMS_ERROR. To get extended error information, call **SmsGetLastError**.

Remarks

The **SmsSendMessage** function is used to send a text message to a mobile device. This API is designed to support multiple methods of sending text messages, with the default method sending the message through a server gateway established by the wireless service provider.

SMS_SERVICE_SMTP

This message service sends the message through the wireless service provider's mail gateway using the SMTP protocol. However, it is important to note that many of these gateways will not accept messages from a client that is connected to them using a residential Internet service provider. If the application is being run on a system that uses a residential provider, that service provider may also block outbound connections to all mail servers other than their own. These anti-spam measures typically require that most end-user applications specify a relay mail server rather than submitting the message directly to the wireless provider's gateway.

Because most wireless carriers in the United States and Canada must provide for wireless number portability, there is the possibility that the provider information returned may no longer correspond to the telephone number. It is recommended that you provide your end-user with the ability to specify an alternate preferred provider to use when sending the text message. For more information, refer to the **SmsGetGateway** function.

This service also sends an HTTP query to the server **api.sockettools.com** to obtain information about the phone number and wireless service provider. This requires that the local system can establish a standard network connection over port 80. If the client cannot connect to the server, the function will fail and an appropriate error will be returned. The server imposes a limit on the

maximum number of connections that can be established and the maximum number of requests that can be issued per minute. If this function is called multiple times over a short period, the library may also force the application to block briefly. Server responses are cached per session, so calling this function multiple times using the same phone number will not increase the request count.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmsGetGateway](#), [SmsGetProvider](#), [SMSMESSAGE](#), [SMSSERVICE](#)

SmsSetLastError Function

```
VOID WINAPI SmsSetLastError(  
    DWORD dwErrorCode  
);
```

The **SmsSetLastError** function sets the error code for the current thread. This function is typically used to clear the last error by passing a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or SMS_ERROR. Those functions which call **SmsSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **SmsGetLastError** function. The use of **SmsGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstxtv10.lib

See Also

[SmsGetErrorString](#), [SmsGetLastError](#)

SmsUninitialize Function

```
VOID WINAPI SmsUninitialize();
```

The **SmsUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **SmsInitialize** call before it can call any of the other the library functions. When it has completed the use of library, the application must call **SmsUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There must be a call to **SmsUninitialize** for every successful call to **SmsInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstxtv10.lib

See Also

[SmsInitialize](#)

Text Message Data Structures

- SMSGATEWAY
- SMSMESSAGE
- SMSPROVIDER
- SMSSERVICE

INITDATA Structure

This structure contains information about the SocketTools library when the initialization function returns.

```
typedef struct _INITDATA
{
    DWORD        dwSize;
    DWORD        dwVersionMajor;
    DWORD        dwVersionMinor;
    DWORD        dwVersionBuild;
    DWORD        dwOptions;
    DWORD_PTR    dwReserved1;
    DWORD_PTR    dwReserved2;
    TCHAR        szDescription[128];
} INITDATA, *LPINITDATA;
```

Members

dwSize

Size of this structure. This structure member must be set to the size of the structure prior to calling the initialization function. Failure to do so will cause the function to fail.

dwVersionMajor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the major version number for the library.

dwVersionMinor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the minor version number for the library.

dwVersionBuild

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the build number for the library.

dwOptions

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved1

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved2

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

szDescription

A null terminated string which describes the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

MSGGATEWAY Structure

This structure contains information about a text message gateway server and the recipient.

```
typedef struct _MSGGATEWAY
{
    INT      nGatewayId;
    INT      nCountryCode;
    INT      nAreaCode;
    INT      nExchange;
    INT      nMessageLength;
    DWORD    dwReserved;
    TCHAR    szProvider[SMS_MAXPROVIDERNAMELEN];
    TCHAR    szDomain[SMS_MAXDOMAINNAMELEN];
    TCHAR    szServer[SMS_MAXMAILSERVERLEN];
    TCHAR    szAddress[SMS_MAXMAILADDRESSLEN];
} SMSCHANNEL, *LPSMSCHANNEL;
```

Members

nGatewayId

An integer value which identifies the gateway record. This value is used internally and an application should not depend on the value not changing over time for a specific telephone number. As new area codes are introduced, the provider database will be updated to reflect these changes and that can result in a change to the gateway ID associated with a specific telephone number.

nCountryCode

An integer value which specifies the ITU country calling code associated with the service provider. Currently this value will always be 1, which is the country code used by North American service providers. If the service provider database is expanded to include additional countries in the future, this value will identify the country of origin.

nAreaCode

An integer value which specifies the Numbering Plan Area (NPA) code, commonly known as the area code. For the United States and Canada, area codes are assigned by the North American Numbering Plan Administration (NANPA). In North America, the area code is digits 1-3 for a 10-digit telephone number. This value, along with the exchange, is used to determine which company provides wireless service for a specific telephone number.

nExchange

An integer value which specifies the exchange area. In North America, the exchange is digits 4-6 for a 10-digit telephone number. This value, along with the area code, is used to determine which company provides wireless service for a specific telephone number.

nMessageLength

An integer value which specifies the maximum number of characters that the service provider will accept for a single text message. If the message exceeds this number of characters, the service provider may reject the message, or it may split the message into multiple messages.

dwReserved

A value reserved for internal use.

szProvider

A pointer to a string which identifies the name of the service provider that is associated with the specified telephone number. Note that this value may not represent the actual company that is

providing the wireless service.

szDomain

A pointer to a string which identifies the gateway domain name used by the service provider to accept text messages for their customer. This domain name is used to determine the actual name of the gateway mail server that is responsible for accepting messages.

szServer

A pointer to a string which identifies the host name or IP address of the mail server used to accept text messages for the specified service provider. In some cases, a provider may have multiple gateway servers and this value will represent the preferred mail server for the domain.

szAddress

A pointer to a string which contains the complete email address that should be used when sending the text message through the gateway mail server. Different service providers can have slightly different rules about how the address is formatted, but it typically is a combination of the telephone number and the gateway domain name.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

SMSMESSAGE Structure

This structure provides information about a text message.

```
typedef struct _SMSMESSAGE
{
    DWORD dwFormat;
    DWORD dwLength;
    DWORD dwFlags;
    DWORD dwReserved;
    LPCTSTR lpszProvider;
    LPCTSTR lpszPhoneNumber;
    LPCTSTR lpszSender;
    LPCTSTR lpszMessage;
} SMSMESSAGE, *LPSMSMESSAGE;
```

Members

dwFormat

An integer value which specifies the format of the message message. This member is included for future use where a service provider supports multiple message formats based on different versions of the protocol. The default value for this member is SMS_FORMAT_TEXT.

dwLength

An integer value which specifies the length of the message. If this member is zero, the length will be automatically calculated based on the length of the *lpszMessage* text that is terminated by a null character. If this value is larger than the actual length of the message text, it will be ignored.

dwFlags

An integer value which specifies one or more message options.

Constant	Description
SMS_MESSAGE_DEFAULT	The default value used with standard text messages.
SMS_MESSAGE_URGENT	The text message should be flagged as urgent. For messages that are sent through a mail gateway, this will set the header to indicate that it is a high priority message. Note that service providers handle urgent messages differently and some may ignore the message priority.

dwReserved

Reserved for future use. This value should always be zero.

lpszProvider

A pointer to a null terminated string which specifies the name of the preferred wireless service provider responsible for handling the message. If this member is NULL or an empty string, the default provider assigned to the recipient's phone number will be used. This structure member is only used with SMS_SERVICE_SMTP messages and is ignored for other message services.

lpszPhoneNumber

A pointer to a null terminated string which specifies the recipient's phone number. This can be a standard E.164 formatted number or an unformatted number. Any extraneous whitespace, punctuation or other non-numeric characters in the string will be ignored. This structure member cannot be NULL.

lpszSender

A pointer to a null terminated string which identifies the sender of the message. For SMS_SERVICE_SMTP messages, this string should be a valid email address. For other services, this string may specify a phone number or shortcode. This structure member cannot be NULL.

lpszMessage

A pointer to a null terminated string that contains the message to be sent to the recipient. In most cases, a message should not exceed 160 characters in length, although some service providers may accept longer messages. If a message exceeds the maximum number of characters accepted by a service provider, the message may be ignored or it may be split into multiple messages.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmsSendMessage](#), [SMSSERVICE](#)

SMSPROVIDER Structure

This structure contains information about a wireless service provider.

```
typedef struct _SMSPROVIDER
{
    INT        nProviderId;
    INT        nCountryCode;
    INT        nRegionCode;
    INT        nMessageLength;
    DWORD      dwFlags;
    DWORD      dwReserved;
    TCHAR      szGuid[SMS_MAXPROVIDERGUIDLEN];
    TCHAR      szName[SMS_MAXPROVIDERNAMELEN];
    TCHAR      szCompany[SMS_MAXCOMPANYNAMELEN];
    TCHAR      szDomain[SMS_MAXDOMAINNAMELEN];
} SMSPROVIDER, *LPSMSPROVIDER;
```

Members

nProviderId

An integer value which identifies the provider record. This value is used internally and an application should not depend on the value not changing over time for a specific service provider. To uniquely identify a provider, use the *szGuid* member of the structure, which is guaranteed not to change as providers are added and removed from the database.

nCountryCode

An integer value which specifies the ITU country calling code associated with the service provider. Currently this value will always be 1, which is the country code used by North American service providers. If the service provider database is expanded to include additional countries in the future, this value will identify the country of origin.

nRegionCode

An integer value which identifies the region that the service provider covers. In North America, each region consists of multiple states and/or provinces. If a provider services multiple regions, this will identify the primary region where they provide coverage.

Constant	Description
SMS_REGION_NATIONAL 0	All regions. This region code is used for service providers that have national coverage and do not exclusively provide service within one or more specific geographical regions.
SMS_REGION_NORTH_EAST_ATLANTIC 1	Northeastern Atlantic region which includes Connecticut, Maine, Massachusetts, New Hampshire, New Brunswick, Newfoundland, Nova Scotia, Rhode Island and Vermont.
SMS_REGION_MIDDLE_ATLANTIC 2	Middle Atlantic region which includes New Jersey, New York, Delaware, District of Columbia, Maryland, Pennsylvania, Quebec, Virginia and West Virginia.
SMS_REGION_EAST_NORTH_CENTRAL 3	Northeastern central region which includes Illinois, Indiana, Michigan, Ohio and

	Wisconsin.
SMS_REGION_SOUTH_ATLANTIC 4	Southern Atlantic region which includes Florida, Georgia, North Carolina and South Carolina.
SMS_REGION_EAST_SOUTH_CENTRAL 5	Southeastern central region which includes Alabama, Kentucky, Mississippi and Tennessee.
SMS_REGION_WEST_NORTH_CENTRAL 6	Northwestern central region which includes Iowa, Kansas, Manitoba, Minnesota, Missouri, Nebraska, North Dakota, Ontario and South Dakota.
SMS_REGION_WEST_SOUTH_CENTRAL 7	Southwestern central region which includes Arkansas, Louisiana, Oklahoma and Texas.
SMS_REGION_MOUNTAIN 8	Mountain region which includes Alberta, Arizona, Colorado, Idaho, Montana, Nevada, New Mexico, Northwest Territories, Saskatchewan, Utah and Wyoming.
SMS_REGION_PACIFIC 9	Pacific region which includes Alaska, British Columbia, California, Hawaii, Oregon, Washington and Yukon.

nMessageLength

An integer value which specifies the maximum number of characters that the service provider will accept for a single text message. If the message exceeds this number of characters, the service provider may reject the message, or it may split the message into multiple messages.

dwFlags

An unsigned integer value which specifies one or more flags that provides additional information about the service provider. This value is constructed by using a bitwise operator with any of the following constants:

Constant	Description
SMS_PROVIDER_DEFAULT 0	A standard service provider. Typically this means that customers have a service contract for their mobile device and pay monthly access and service charges.
SMS_PROVIDER_PREPAID 1	A service provider that offers pre-paid calling cards or fixed month-to-month payments that do not require long-term service contracts.

dwReserved

A value reserved for internal use.

szGuid

A pointer to a string which uniquely identifies the service provider. The string is in a standard format used for globally unique identifiers (GUIDs) and is guaranteed to not change for the service provider it has been assigned to.

szName

A pointer to a string which specifies the name of service provider. Note that this value may not represent the actual company that is providing the wireless service.

szCompany

A pointer to a string which specifies the name of the company associated with the service provider. This may be the same as name of the service provider itself or it may be the name of a parent company that owns the service provider.

szDomain

A pointer to a string which identifies the gateway domain name used by the service provider to accept text messages for their customer. This domain name is used to determine the actual name of the gateway mail server that is responsible for accepting messages.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

SMSSERVICE Structure

This structure provides information about the service used to send a text message.

```
typedef struct _SMSSERVICE
{
    DWORD dwServiceType;
    DWORD dwAuthType;
    DWORD dwVersion;
    DWORD dwTimeout;
    DWORD dwOptions;
    DWORD dwReserved;
    LPCTSTR lpszResource;
    LPCTSTR lpszAccount;
    LPCTSTR lpszUserName;
    LPCTSTR lpszPassword;
} SMSSERVICE, *LPSMSSERVICE;
```

Members

dwServiceType

An integer value which identifies the type of service that will be used to send the message. This member can be one of the following values:

Constant	Description
SMS_SERVICE_SMTP	The text message will be sent through the mail gateway for the specified service provider. This service uses SMTP to submit the message for delivery, either directly to the server provider's mail gateway server or through a relay server. This is the default service type.

dwAuthType

An integer value which identifies the type of authentication used with the service. This member can be one of the following values:

Constant	Description
SMS_AUTH_DEFAULT	The default authentication method for this service type should be used. Most applications should use this value unless a service type provides multiple authentication methods.
SMS_AUTH_USERNAME	The service requires authentication using a username and password. This value can be used with an SMTP service that requires user authentication and is typically needed when using a mail server relay. For the SMS_SERVICE_SMTP service, this is the default authentication method.

dwVersion

An integer value which identifies the interface version for the service being used. This member is included for future use where a service may support multiple versions of their interface and should normally be set to the value SMS_VERSION_DEFAULT.

dwTimeout

An integer value which specifies the amount of time in seconds that a function will wait for a

response from the service. If this value is zero, a default timeout period of 20 seconds will be used. If the service does not respond within this time period, the function will fail.

dwOptions

An integer value which specifies one or more options.

Constant	Description
SMS_OPTION_NONE	No additional options for the service.
SMS_OPTION_SECURE	This option specifies that SSL/TLS will be used to establish a secure, encrypted connection with the service. For some services, it may be required to connect to them securely and this option will be enabled automatically.

dwReserved

Reserved for future use. This value should always be zero.

lpszResource

A pointer to a null terminated string that specifies a resource for the service. Typically this will be either a fully qualified domain name or a URL. For gateways using SMTP, this string should identify the mail server. An alternate port number can also be specified by appending it to the hostname, separated by a colon. For example, **smtp.company.com:587** would connect to the server on port 587. If you are specifying an IPv6 address with an alternate port number, the address must be enclosed in brackets. For services where a domain name or resource URL is not required, this member will be ignored and can be NULL.

lpszAccount

A pointer to a null terminated string that specifies an account name or identifier. Some service providers may require a unique account name or a token (application ID) in conjunction with other credentials. This member is not used with mail gateways and ignored if the service type is SMS_SERVICE_SMTP. If no account name is required for session authentication, this member can be NULL.

lpszUserName

A pointer to a null terminated string that specifies a user name to authenticate the session. If the authentication type is SMS_AUTH_USERNAME this member must specify a valid user name. If no authentication is required, this member may be NULL. Note that some service providers may use terminology other than "username" with their documentation; this member will always specify the first of a pair of authentication tokens.

lpszPassword

A pointer to a null terminated string that specifies the password used to authenticate the session. If the authentication type is SMS_AUTH_USERNAME this member must specify a valid password. If no authentication is required, this member may be NULL. Note that some service providers may use terminology other than "password" with their documentation; this member will always specify the second of a pair of authentication tokens.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Unicode: Implemented as Unicode and ANSI versions.

See Also

[SmsSendMessage](#), [SMSMESSAGE](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

Time Protocol Library

Query a time server for the current time and synchronize the local system clock with that value.

Reference

- [Functions](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

File Name	CSTIMV10.DLL
Version	10.0.1468.2518
LibID	7E65DD9F-5799-42FA-A72D-EC0B714E3021
Import Library	CSTIMV10.LIB
Dependencies	None
Standards	RFC 868

Overview

The Time Protocol library provides an interface for synchronizing the local system's time and date with that of a server. The time values returned are in Coordinated Universal Time and be adjusted for the local host's timezone. The library enables developers to query a server for the current time and then update the system clock if desired.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Time Protocol Functions

Function	Description
ConvertNetworkTime	Convert network time to system time
ConvertSystemTime	Convert system time to network time
GetNetworkTime	Return the current network time from the server
QueryNetworkTime	Query the server for the network time
ReadNetworkTime	Read the network time returned by the server
TimeDisableTrace	Disable logging of socket function calls to the trace log
TimeEnableTrace	Enable logging of socket function calls to a file
TimeGetErrorString	Return a description for the specified error code
TimeGetLastError	Return the last error code
TimeInitialize	Initialize the library and validate the specified license key at runtime
TimeSetLastError	Set the last error code
TimeUninitialize	Terminate use of the library by the application
UpdateLocalTime	Update the local system time with the network time

ConvertNetworkTime Function

```
BOOL WINAPI ConvertNetworkTime(  
    DWORD dwNetworkTime,  
    LPSYSTEMTIME lpSystemTime,  
    BOOL bLocalTime  
);
```

The **ConvertNetworkTime** function converts the specified network time, adjusting for the local timezone if required. The network time is a 32-bit number, represented as the number of seconds since midnight, 1 January 1900 UTC.

Parameters

dwNetworkTime

The network time to be converted.

lpSystemTime

A pointer to a [SYSTEMTIME](#) structure which will be modified for the specified network time.

bLocalTime

A boolean flag that is used to specify if the network time should be adjusted for the local timezone.

Return Value

If the network time could be converted, the function returns a non-zero value. If the network time cannot be converted, or the pointer to the SYSTEMTIME structure is invalid, the function will return zero.

Remarks

The network time value can represent a date and time up to the year 2036.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstimv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ConvertSystemTime](#)

ConvertSystemTime Function

```
DWORD WINAPI ConvertSystemTime(  
    LPSYSTEMTIME lpSystemTime  
);
```

The **ConvertSystemTime** function converts the specified system time to network time. The network time value is a 32-bit number, represented as the number of seconds since midnight, 1 January 1900 UTC.

Parameters

lpSystemTime

A pointer to a [SYSTEMTIME](#) structure which will be modified for the specified network time.

Return Value

If the system time could be converted, the function returns the number of seconds since midnight, 1 January 1900 UTC. If the pointer to the [SYSTEMTIME](#) structure is invalid, or the structure contains invalid data, the function will return zero.

Remarks

The network time value can represent a date and time up to the year 2036.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstimv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ConvertNetworkTime](#), [UpdateLocalTime](#)

GetNetworkTime Function

```
DWORD WINAPI GetNetworkTime(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout  
);
```

The **GetNetworkTime** function returns the network time from the specified host. The network time is a 32-bit number, represented as the number of seconds since midnight, 1 January 1900 UTC.

Parameters

lpszRemoteHost

A pointer to the name of the server. The host must be running a time server that complies with the specifications outlined in RFC 868.

nRemotePort

The port the time server is running on. A value of zero indicates that the default port number for the service should be used.

nTimeout

The number of seconds that the function will wait for a response from the server.

Return Value

If the function succeeds, it returns the number of seconds since midnight, 1 January 1900 UTC. If the function was unable to obtain the time from the specified host, it returns zero.

Remarks

The **GetNetworkTime** function will cause the calling thread to block until the time is returned by the server, or the operation times out. For applications which require asynchronous operation, the **QueryNetworkTime** function should be used instead.

The network time value can represent a date and time up to the year 2036. It is important to note that the network time value is not the same as the UNIX time value that is used the standard C library time functions.

In the United States, the National Institute of Standards and Technology (NIST) hosts a number of public servers which can be used to obtain the current time. The following table lists the current host names and addresses:

Server Name	IP Address	Location
time-a.nist.gov	129.6.15.28	Gaithersburg, Maryland
time-b.nist.gov	129.6.15.29	Gaithersburg, Maryland
time-nw.nist.gov	131.107.13.100	Redmond, Washington
time-a.timefreq.bldrdoc.gov	132.163.4.101	Boulder, Colorado
time-b.timefreq.bldrdoc.gov	132.163.4.102	Boulder, Colorado
time-c.timefreq.bldrdoc.gov	132.163.4.103	Boulder, Colorado

Time servers are also commonly maintained by Internet service providers and universities. If you are unable to obtain the time from a server, contact the system administrator to determine if they

have the standard time service available on port 37.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstimv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ConvertNetworkTime](#), [QueryNetworkTime](#), [ReadNetworkTime](#), [TimeInitialize](#)

QueryNetworkTime Function

```
HCLIENT WINAPI QueryNetworkTime(  
    LPCTSTR LpszRemoteHost,  
    UINT nRemotePort,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **QueryNetworkTime** function connects to the specified server and issues a request for the current network time. The handle is passed as a parameter to the **ReadNetworkTime** function to read the value returned by the server.

Parameters

LpszRemoteHost

A pointer to the name of the server. The host must be running a time server that complies with the specifications outlined in RFC 868.

nRemotePort

The port the time server is running on. A value of zero indicates that the default port number for the service (37) should be used.

hEventWnd

A handle to the window that will receive the notification that the network time has been returned by the server.

uEventMsg

The notification message that will be sent to the window, indicating that the client has received the network time from the server.

Return Value

If the function succeeds, it returns a handle which can be used with the obtain the network time. If the function fails, it returns INVALID_CLIENT. To get extended error information, call **TimeGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
TIME_EVENT_QUERY	The connection to the server has completed and the request for the time has been sent. The client should wait for the response from the server.
TIME_EVENT_REPLY	The server has replied to the request for the time. The client should call the ReadNetworkTime function to read the value returned by the server.
TIME_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.

TIME_EVENT_CANCEL

The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cstimv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[ConvertNetworkTime](#), [GetNetworkTime](#), [ReadNetworkTime](#), [UpdateLocalTime](#)

ReadNetworkTime Function

```
DWORD WINAPI ReadNetworkTime(  
    HCLIENT hClient  
);
```

The **ReadNetworkTime** function returns the network time from the specified host.

Parameters

hClient

The client handle.

Return Value

If the function succeeds, it returns the number of seconds since midnight, 1 January 1900 UTC. If the function was unable to read the time from the specified host, it returns zero.

Remarks

The **ReadNetworkTime** function reads the time value returned by the server using the handle returned by the **QueryNetworkTime** function. The network time value can represent a date and time up to the year 2036.

To convert the time value to a SYSTEMTIME structure, use the **ConvertNetworkTime** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cstimv10.lib

See Also

[ConvertNetworkTime](#), [QueryNetworkTime](#)

TimeAttachThread Function

```
DWORD WINAPI TimeAttachThread(  
    HCLIENT hClient  
    DWORD dwThreadId  
);
```

The **TimeAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is `TIME_ERROR`. To get extended error information, call **TimeGetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **TimeAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **TimeAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **TimeCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **TimeUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstimv10.lib`

See Also

[QueryNetworkTime](#), [ReadNetworkTime](#), [TimeInitialize](#), [TimeUninitialize](#)

TimeDisableEvents Function

```
INT WINAPI TimeDisableEvents(  
    HCLIENT hClient  
);
```

The **TimeDisableEvents** function disables all event notification, including event callbacks. Any pending events are deleted.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TIME_ERROR. To get extended error information, call **TimeGetLastError**.

Remarks

This function affects both event notification and event callbacks. Any outstanding events in the message queue should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstimv10.lib

See Also

[TimeEnableEvents](#), [TimeFreezeEvents](#), [TimeRegisterEvent](#)

TimeDisableTrace Function

```
BOOL WINAPI TimeDisableTrace();
```

The **TimeDisableTrace** function disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstimv10.lib

See Also

[TimeEnableTrace](#)

TimeEnableEvents Function

```
INT WINAPI TimeEnableEvents(  
    HCLIENT hClient,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **TimeEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and is retained for backwards compatibility. Applications should use the **TimeRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the event notification window. This window receives a user-defined message which specifies the event that has occurred. If this value is NULL, event notification is disabled.

uEventMsg

An unsigned integer which specifies the user-defined message that is sent when an event occurs. This parameter's value must be greater than the value of WM_USER.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is TIME_ERROR. To get extended error information, call **TimeGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
TIME_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
TIME_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
TIME_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
TIME_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.

TIME_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
TIME_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To cancel asynchronous notification and return the client to a blocking mode, use the **TimeDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cstimv10.lib

See Also

[TimeDisableEvents](#), [TimeFreezeEvents](#), [TimeRegisterEvent](#)

TimeEnableTrace Function

```
BOOL WINAPI TimeEnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **TimeEnableTrace** function enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace function logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstimv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TimeDisableTrace](#)

TimeEventProc Function

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

The **TimeEventProc** function is an application-defined callback function that processes events generated by the calling process.

Parameters

hClient

The handle to the client session.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **TimeRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **TimeRegisterEvent** function. The **TimeEventProc** function is a placeholder for the application-defined function name.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cstimv10.lib

See Also

[TimeDisableEvents](#), [TimeEnableEvents](#), [TimeFreezeEvents](#), [TimeRegisterEvent](#)

TimeFreezeEvents Function

```
INT WINAPI TimeFreezeEvents(  
    HCLIENT hClient,  
    BOOL bFreeze  
);
```

The **TimeFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `TIME_ERROR`. To get extended error information, call **TimeGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstimv10.lib`

See Also

[TimeDisableEvents](#), [TimeEnableEvents](#), [TimeRegisterEvent](#)

TimeGetErrorString Function

```
INT WINAPI TimeGetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);
```

The **TimeGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **TimeGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The last-error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is 0, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstimv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[TimeGetLastError](#), [TimeSetLastError](#)

TimeGetLastError Function

```
DWORD WINAPI TimeGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **TimeSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **TimeGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **TimeSetLastError(0)** when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or TIME_ERROR. Those functions which call **TimeSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cstimv10.lib

See Also

[TimeGetErrorString](#), [TimeSetLastError](#)

TimeGetStatus Function

```
INT WINAPI TimeGetStatus(  
    HCLIENT hClient  
);
```

The **TimeGetStatus** function returns the current status of the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is TIME_ERROR. To get extended error information, call **TimeGetLastError**.

Remarks

The **TimeGetStatus** function returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Constant	Description
1	TIME_STATUS_IDLE	The client is current idle and not sending or receiving data.
2	TIME_STATUS_CONNECT	The client is establishing a connection with the server.
3	TIME_STATUS_READ	The client is reading data from the server.
4	TIME_STATUS_WRITE	The client is writing data to the server.
5	TIME_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstimv10.lib

See Also

[QueryNetworkTime](#), [ReadNetworkTime](#)

TimeInitialize Function

```
BOOL WINAPI TimeInitialize(  
    LPCTSTR lpszLicenseKey,  
    LPINITDATA lpData  
);
```

The **TimeInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpData

Pointer to an INITDATA data structure. This parameter may be NULL if the initialization data for the library is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **TimeGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must insure that this function is called with a valid license key.

If the *lpData* argument is specified, it must point to an INITDATA structure which has been initialized by setting the *dwSize* member to the size of the structure. All other structure members should be set to zero. If the function is successful, then the INITDATA structure will be filled with identifying information about the library.

Although it is only required that **TimeInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **TimeUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **TimeInitialize** or **TimeUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstimv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

TimeRegisterEvent Function

```
INT WINAPI TimeRegisterEvent(  
    HCLIENT hClient,  
    UINT nEventId,  
    INETEVENTPROC LpfnEvent,  
    DWORD_PTR dwParam  
);
```

The **TimeRegisterEvent** function registers a callback function for the specified event.

Parameters

hClient

Handle to client session.

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One or more of the following values may be used:

Constant	Description
TIME_EVENT_CONNECT	The connection to the server has completed.
TIME_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
TIME_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
TIME_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
TIME_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
TIME_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

lpfnEvent

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **TimeEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the

x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `TIME_ERROR`. To get extended error information, call **TimeGetLastError**.

Remarks

The **TimeRegisterEvent** function associates a callback function with a specific event. The event handler is an **TimeEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the *lpEventProc* parameter must be declared using the `__stdcall` calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cstimv10.lib`

See Also

[TimeDisableEvents](#), [TimeEnableEvents](#), [TimeEventProc](#), [TimeFreezeEvents](#)

TimeSetLastError Function

```
VOID WINAPI TimeSetLastError(  
    DWORD dwErrorCode  
);
```

The **TimeSetLastError** function sets the error code for the current thread. This function is typically used to clear the last error by passing a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or TIME_ERROR. Those functions which call **TimeSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **TimeGetLastError** function. The use of **TimeGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cstimv10.lib

See Also

[TimeGetErrorString](#), [TimeGetLastError](#)

TimeUninitialize Function

```
VOID WINAPI TimeUninitialize();
```

The **TimeUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **TimeInitialize** call before it can call any of the other the library functions. When it has completed the use of library, the application must call **TimeUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There must be a call to **TimeUninitialize** for every successful call to **TimeInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cstimv10.lib

See Also

[TimeInitialize](#)

UpdateLocalTime Function

```
BOOL WINAPI UpdateLocalTime(  
    DWORD dwNetworkTime  
);
```

The **UpdateLocalTime** function sets the local system clock to the specified date and time.

Parameters

dwNetworkTime

The date and time the system clock should be set to, represented as the number of seconds since midnight, 1 January 1900.

Return Value

If the function is able to update the local time, it returns a non-zero value. If the specified time is invalid, or the user does not have the access rights to change the system clock, the function returns zero.

Remarks

The network time value can represent a date and time up to the year 2036.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cstimv10.lib

See Also

[ConvertNetworkTime](#), [GetNetworkTime](#), [QueryNetworkTime](#), [TimeInitialize](#)

Time Protocol Data Structures

- INITDATA
- SYSTEMTIME

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

INITDATA Structure

This structure contains information about the SocketTools library when the initialization function returns.

```
typedef struct _INITDATA
{
    DWORD        dwSize;
    DWORD        dwVersionMajor;
    DWORD        dwVersionMinor;
    DWORD        dwVersionBuild;
    DWORD        dwOptions;
    DWORD_PTR    dwReserved1;
    DWORD_PTR    dwReserved2;
    TCHAR        szDescription[128];
} INITDATA, *LPINITDATA;
```

Members

dwSize

Size of this structure. This structure member must be set to the size of the structure prior to calling the initialization function. Failure to do so will cause the function to fail.

dwVersionMajor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the major version number for the library.

dwVersionMinor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the minor version number for the library.

dwVersionBuild

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the build number for the library.

dwOptions

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved1

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved2

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

szDescription

A null terminated string which describes the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

SYSTEMTIME Structure

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include windows.h.

Web Services Library

The Web Services library provides data storage and location services for applications.

Reference

- [Functions](#)
- [Constants](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

File Name	CSWEBV10.DLL
Version	10.0.1468.2518
LibID	62AB4693-FFA1-4A6F-BC44-FDC113EB2C68
Import Library	CSWEBV10.LIB
Dependencies	None

Overview

The Web Services library enables an application to store and manage data remotely, and return information about the current physical location of the client. These functions use secure services provided by SocketTools API servers and do not require third-party APIs or accounts with other cloud service providers.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Web Services Functions

Function	Description
WebCancelTransfer	Cancel a storage operation that is currently in progress
WebCloseStorage	Close a storage container and release resources allocated for the client session
WebCompareFile	Compare the contents of a stored object with a local file
WebCompareObject	Compare the contents of a stored object with a memory buffer
WebCompareText	Compare the contents of a stored object with a string
WebCopyObject	Copy a storage object to a new location and optionally rename the object label
WebDeleteObject	Delete an existing storage object
WebDownloadFile	Download the contents of a stored object to a local file
WebDownloadFileEx	Download the contents of a stored object with progress notifications
WebDisableTrace	Disable logging of function calls to the trace log
WebEnableTrace	Enable logging of function calls to a file
WebEnumApplications	Enumerate all registered storage applications
WebEnumObjects	Enumerate all storage objects that match the specified label or content type
WebGetAccountId	Return the current web services account identifier
WebGetErrorString	Return a description for the specified error code
WebGetFile	Download the contents of a storage object and copy to a local file
WebGetFileEx	Download the contents of a storage object and return information about the object
WebGetFirstApplication	Return information about the first registered storage application
WebGetFirstObject	Return information about the first object that matches search criteria
WebGetLastError	Return the last error code
WebGetLocation	Return the current physical location of the local computer system
WebGetNextApplication	Return information about the next registered storage application
WebGetNextObject	Return information about the next object that matches search criteria
WebGetObject	Download the contents of a storage object to a buffer
WebGetObjectEx	Download the contents of a storage object to buffer and return information about the object
WebGetObjectInformation	Retrieve the metadata for the specified storage object
WebGetObjectSize	Return the size of the specified storage object
WebGetStorageId	Return the returns the storage container ID
WebGetStorageQuota	Return quota limits assigned to your storage account
WebGetStorageTimeout	Get the number of seconds until a storage operation times out
WebGetTextObject	Download the contents of a text object into a string buffer
WebGetTransferStatus	Return status information about the progress of a data transfer
WebInitialize	Initialize the library and validate the specified license key at runtime
WebMoveObject	Move a storage object to a new location and optionally rename the object label
WebOpenStorage	Open a storage container and return a handle for the client session
WebPutFile	Upload the contents of a local file to a new storage object

WebPutFileEx	Upload the contents of a local file and return information about the new object
WebPutObject	Upload the contents of a memory buffer to a new storage object
WebPutObjectEx	Upload the contents of a memory buffer and return information about the new object
WebPutTextObject	Create or replace a text object with the contents of a string buffer
WebRegisterAppId	Register a new application identifier used to store and retrieve data
WebRegisterEvent	Register an event handler to receive notifications for the session
WebRenameObject	Change the label associated with a storage object
WebResetStorage	Resets the application storage container and deletes all stored objects
WebResetStorageEx	Resets the specified storage container for an application and deletes all stored objects
WebSetLastError	Set the last error code
WebSetStorageTimeout	Set the number of seconds until a storage operation times out
WebUninitialize	Terminates the use of the library
WebUnregisterAppId	Unregister the application identifier and delete all associated storage objects
WebUploadFile	Upload the contents of a local file, creating or overwriting a stored object
WebUploadFileEx	Upload the contents of a local file with progress notifications
WebValidateAppId	Validate the specified application identifier
WebValidateLabel	Check the specified string to ensure it is a valid object label

WebCancelTransfer Function

```
BOOL WINAPI WebCancelTransfer(  
    HSTORAGE hStorage  
);
```

The **WebCancelTransfer** function cancels the current data transfer in progress.

Parameters

hStorage

A handle to the storage container.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebCancelTransfer** function will cancel the current data transfer and abort the connection to the storage server. This function will fail if an active data transfer (either an upload or download) is not in progress.

This would typically be used within an event handler to cancel an operation as the contents of an object are being read or written. There is no mechanism to resume a canceled data transfer and this function should only be used when absolutely necessary.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswebv10.lib

See Also

[WebGetTransferStatus](#), [WebRegisterEvent](#)

WebCloseStorage Function

```
BOOL WINAPI WebCloseStorage(  
    HSTORAGE hStorage  
);
```

The **WebCloseStorage** function closes the storage container.

Parameters

hStorage

A handle to the storage container.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebCloseStorage** function must be called after all operations using the storage container have completed. The access token granted to the application will be released and the memory allocated for the session cache will be freed. Failure to call this function can result in a memory leak.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

See Also

[WebOpenStorage](#)

WebCompareFile Function

```
BOOL WINAPI WebCompareFile(  
    HSTORAGE hStorage,  
    LPCTSTR lpszObjectLabel,  
    LPCTSTR lpszLocalFile  
);
```

The **WebCompareFile** function compares the contents of a stored object with a local file.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A pointer to a null terminated string which specifies the label of the object to be compared.

lpszLocalFile

A pointer to a null terminated string that specifies the name of the local file. If a path is not specified, the file will be created in the current working directory.

Return Value

If the function succeeds, the return value is a non-zero and the contents of the file matches the stored object. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebCompareFile** function performs a binary comparison of the contents of a local file with a stored object on the server. The contents of the file must be identical to the contents of the stored object or the function will fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebCompareObject](#), [WebCompareText](#), [WebGetObjectInformation](#)

WebCompareObject Function

```
BOOL WINAPI WebCompareObject(  
    HSTORAGE hStorage,  
    LPCTSTR lpszObjectLabel,  
    LPCVOID lpvBuffer,  
    DWORD dwLength  
);
```

The **WebCompareObject** function compares the contents of a stored object with the data provided by the caller.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A pointer to a null terminated string which specifies the label of the object to be compared.

lpvBuffer

A pointer to a buffer that contains the data to be compared against the storage object.

dwLength

An unsigned integer which specifies the length of the data buffer to be compared.

Return Value

If the function succeeds, the return value is a non-zero and the data matches the contents of the stored object. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebCompareObject** function performs a binary comparison of the data in the specified buffer with the contents of the storage object on the server. The *dwLength* parameter must match the size of the stored object exactly, or this function will fail. Partial comparisons are not supported by this function.

If you wish to compare the contents of a text object, it is recommended that you use **WebCompareText**. This function ensures that Unicode text is compared correctly.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebCompareFile](#), [WebCompareText](#), [WebGetObjectInformation](#)

WebCompareText Function

```
BOOL WINAPI WebCompareText(  
    HSTORAGE hStorage,  
    LPCTSTR lpszObjectLabel,  
    LPCTSTR lpszObjectText,  
    INT cchObjectText  
);
```

The **WebCompareText** function compares the contents of a stored object with the string provided by the caller.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A pointer to a null terminated string which specifies the label of the object to be compared.

lpszObjectText

A pointer to a null terminated string which contains the text to be compared with the stored object.

cchObjectText

The number of characters in the *lpszObjectText* string to be compared. This value may be -1, in which case the string length will be determined by counting the number of characters up to the terminating null.

Return Value

If the function succeeds, the return value is a non-zero and the data matches the contents of the stored object. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebCompareText** function performs a text comparison of the characters in the string with the contents of the storage object on the server. The string length must match the amount of text in the stored object exactly, or this function will fail. Partial comparisons are not supported by this function.

The Unicode version of this function will automatically convert the UTF-16 string to UTF-8 encoding in the same way the **WebPutTextObject** function does.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebCompareFile](#), [WebCompareObject](#), [WebGetObjectInformation](#), [WebGetTextObject](#), [WebPutTextObject](#)

WebCopyObject Function

```
BOOL WINAPI WebCopyObject(  
    HSTORAGE hStorage,  
    LPCTSTR lpszOldLabel,  
    LPCTSTR lpszNewLabel,  
    DWORD dwStorageType,  
    LPWEB_STORAGE_OBJECT lpObject  
);
```

The **WebCopyObject** function creates a copy of an existing storage object using a new label.

Parameters

hStorage

A handle to the storage container.

lpszOldLabel

A pointer to a null terminated string which specifies the label of the existing storage object to be copied. This parameter must specify a valid object label and cannot be a NULL pointer or an empty string.

lpszNewLabel

A pointer to a null terminated string which specifies the name of the new storage object that will be created. This parameter may be NULL or point to an empty string, in which case the label name is not changed. In this case, the *dwStorageType* parameter cannot be WEB_STORAGE_DEFAULT.

dwStorageType

An integer value that identifies the storage container type. One of the following values should be specified:

Constant	Description
WEB_STORAGE_DEFAULT (0)	The default storage type. If this value is specified, the new object will be created using the same storage type as the original storage object.
WEB_STORAGE_GLOBAL (1)	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN (2)	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
WEB_STORAGE_MACHINE (3)	Local machine storage. Objects stored using this storage type are only available to users on the same local

	machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER (4)	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain information about the new storage object. If this information is not required, this parameter may be a NULL pointer.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebCopyObject** function is used to create a copy of an existing storage object. It may be used to duplicate an object with a different label, or it may be used to copy the object to a new storage container type. For example, it can copy an object originally created using WEB_STORAGE_USER to a new object stored using WEB_STORAGE_MACHINE.

Copied objects are assigned their own unique ID and are not linked to one another. Any subsequent changes made to the original object will not affect the copied object. Attempting to copy an object to itself or another existing object will result in an error.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebDeleteObject](#), [WebMoveObject](#), [WebRenameObject](#)

WebDeleteObject Function

```
BOOL WINAPI WebDeleteObject(  
    HSTORAGE hStorage,  
    LPCTSTR lpszObjectLabel  
);
```

The **WebDeleteObject** function deletes an object from the storage container.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A pointer to a null terminated string which specifies the label of the object to be deleted.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebDeleteObject** function permanently deletes the storage object and its associated data from the server. Deleted objects cannot be recovered by the application. To remove all objects stored in the container, use the **WebResetStorage** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebCopyObject](#), [WebMoveObject](#), [WebRenameObject](#), [WebResetStorage](#)

WebDisableTrace Function

```
BOOL WINAPI WebDisableTrace();
```

The **WebDisableTrace** function disables the logging of socket function calls to the trace log.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, a value of zero is returned.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

See Also

[WebEnableTrace](#)

WebDownloadFile Function

```
BOOL WINAPI WebDownloadFile(  
    LPCTSTR lpzLocalFile,  
    LPCTSTR lpzObjectLabel,  
    LPWEB_STORAGE_OBJECT lpObject  
);
```

The **WebDownloadFile** function downloads the contents of a storage object and copies it to a local file.

Parameters

lpzObjectLabel

A pointer to a null terminated string that specifies the label of the object that should be retrieved from the server. This parameter cannot be NULL or an empty string. Refer to the **WebValidateLabel** function for more information about object labels.

lpzLocalFile

A pointer to a null terminated string that specifies the name of the local file that will be created or overwritten with the contents of the storage object. If a path is not specified, the file will be created in the current working directory.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain additional information about the object that has been downloaded. This parameter may be NULL if the information is not required.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebDownloadFile** function downloads the contents of the storage object and stores it in a local file. The object must have been previously created in the storage container `WEB_STORAGE_GLOBAL` using the application ID **SocketTools.Storage.Default**. This is the same default container used by the **WebUploadFile** function.

Unlike the **WebGetFile** and **WebGetFileEx** functions, it is not required that you open a storage container using the **WebOpenStorage** function prior to calling **WebDownloadFile**.

Additional metadata about the object will be returned in the `WEB_STORAGE_OBJECT` structure provided by the caller, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents.

The **WebDownloadFileEx** function provides similar functionality with a more complex interface that supports custom application IDs, additional options and the ability to provide a callback function that is invoked during the download process.

Example

```
WEB_STORAGE_OBJECT webObject;  
  
// Download the object from global storage to a local file  
if (WebDownloadFile(lpzLocalFile, lpzObjectLabel, &webObject))  
{
```

```
// The object was downloaded, display the metadata
_tprintf(_T("Object:  %s\n"), webObject.szObjectId);
_tprintf(_T("Label:   %s\n"), webObject.szLabel);
_tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
_tprintf(_T("Digest:  %s\n"), webObject.szDigest);
_tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The object could not be downloaded, display the error
    TCHAR szError[128];

    WebGetErrorString(WebGetLastError(), szError, 128);
    _tprintf(_T("Unable to download \"%s\" (%s)\n"), lpszObjectLabel, szError);
}
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebDownloadFileEx](#), [WebGetFile](#), [WebUploadFile](#), [WebValidateLabel](#)

WebDownloadFileEx Function

```
BOOL WINAPI WebDownloadFileEx(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszObjectLabel,  
    LPCTSTR lpszAppId,  
    DWORD dwStorageType,  
    DWORD dwReserved,  
    DWORD dwTimeout,  
    LPWEB_STORAGE_OBJECT LpObject,  
    WEBEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

The **WebDownloadFileEx** function downloads the contents of a storage object and copies it to a local file.

Parameters

lpszLocalFile

A pointer to a null terminated string that specifies the name of the local file that will be created or overwritten with the contents of the storage object. If a path is not specified, the file will be created in the current working directory.

lpszObjectLabel

A pointer to a null terminated string that specifies the label of the object that should be retrieved from the server.

lpszAppId

A pointer to null terminated string which specifies the application ID for the storage container. The application ID is a string that uniquely identifies the application and can only contain letters, numbers, the period and the underscore character. If this parameter is NULL or an empty string, the default identifier **SocketTools.Storage.Default** will be used.

dwStorageType

An integer value that identifies the storage container type. One of the following values should be specified:

Constant	Description
WEB_STORAGE_GLOBAL (1)	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN (2)	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.

WEB_STORAGE_MACHINE (3)	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER (4)	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

dwReserved

An unsigned integer value that is reserved for future use. This value should always be zero.

dwTimeout

An unsigned integer value that specifies a timeout period in seconds. If this value is zero, a default timeout period will be used.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain additional information about the object that has been downloaded. This parameter may be NULL if the information is not required.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **WebEventProc** callback function. If this parameter is NULL, no callback function will be invoked during the data transfer.

dwParam

A user-defined integer value that is passed to the callback function specified by *lpEventProc*. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebDownloadFileEx** function downloads the contents of the storage object and stores it in a local file. The **WebDownloadFile** function provides a simpler interface that defaults to downloading an object from the global storage container.

Unlike the **WebGetFile** and **WebGetFileEx** functions, it is not required that you open a storage container using the **WebOpenStorage** function prior to calling **WebDownloadFileEx**.

Additional metadata about the object will be returned in the [WEB_STORAGE_OBJECT](#) structure provided by the caller, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents.

If you are downloading a large object and want your application to receive progress updates during the data transfer, provide a pointer to a callback function as the *lpEventProc* parameter. That function will receive event notifications as the data is being downloaded.

Example

```
WEB_STORAGE_OBJECT webObject;

// Download the object from global storage to a local file
if (WebDownloadFileEx(lpszLocalFile,
                    lpszObjectLabel,
                    lpszAppId,
                    WEB_STORAGE_GLOBAL,
                    0,
                    &webObject,
                    NULL, 0))
{
    // The object was downloaded, display the metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The object could not be retrieved, display the error
    TCHAR szError[128];

    WebGetErrorString(WebGetLastError(), szError, 128);
    _tprintf(_T("Unable to retrieve \"%s\" (%s)\n"), lpszObjectLabel, szError);
}
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebDownloadFile](#), [WebGetFileEx](#), [WebPutFileEx](#), [WebUploadFile](#), [WebUploadFileEx](#)

WebEnableTrace Function

```
BOOL WINAPI WebEnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **WebEnableTrace** function enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace function logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebDisableTrace](#)

WebEnumApplications Function

```
BOOL WINAPI WebEnumApplications(  
    LPWEB_STORAGE_APPLICATION LpApplications,  
    LPDWORD LpdwRegistered  
);
```

The **WebEnumApplications** function enumerates all registered applications associated with the storage account.

Parameters

hStorage

lpApplications

A pointer to an array of [WEB_STORAGE_APPLICATION](#) structures that will contain information about the enumerated application IDs. If this parameter is NULL, then no information is returned.

lpdwRegistered

A pointer to an unsigned integer that will contain the number of registered applications enumerated by this function. If the *lpApplications* parameter points to an array of [WEB_STORAGE_APPLICATION](#) structures, this parameter must be initialized to the maximum size of the array being passed to the function. If the *lpApplications* parameter is NULL, this value must be initialized to zero and when the function returns it will contain the number of registered applications.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebEnumApplications** function can be used to enumerate the application IDs registered with the storage account using the **WebRegisterAppId** function.

This function can be used in two ways. If the *lpApplications* parameter is NULL and the value passed by reference in the *lpdwRegistered* parameter is initialized to zero, the function will return the number of registered applications in *lpdwRegistered*. This can be used to dynamically determine the size of the *lpApplications* array instead of declaring a fixed-size array in your application.

If the *lpApplications* parameter is not NULL, then the value referenced by *lpdwRegistered* must be initialized to the maximum size of the array that *lpApplications* points to. It is important to note that if either parameter is not initialized correctly, it can result in memory corruption and/or an unhandled exception.

The **WebGetFirstApplication** and **WebGetNextApplication** functions can be used to iterate through all registered application IDs without requiring you to preallocate memory for an array. This can be more efficient if a large number of application IDs have been registered.

Example

```
DWORD dwAppCount = 0;  
  
if (WebEnumApplications(NULL, &dwAppCount ))  
{
```

```

LPWEB_STORAGE_APPLICATION lpAppArray = NULL;
DWORD dwIndex;

// Return if there are no registered applications
if (dwAppCount == 0)
    return;

// Allocate memory for an array of WEB_STORAGE_APPLICATION structures
lpAppArray = (LPWEB_STORAGE_APPLICATION)LocalAlloc(LPTR,
sizeof(WEB_STORAGE_APPLICATION) * dwAppCount);

if (lpAppArray == NULL)
    return; // Exhausted virtual memory?

// Enumerate the registered application IDs
if (!WebEnumApplications(lpAppArray, &dwAppCount))
    return;

// Print information about each application
for (dwIndex = 0; dwIndex < dwAppCount ; dwIndex++)
{
    _tprintf(_T("AppId: %s\n"), lpAppArray[dwIndex].szAppId);
    _tprintf(_T("Key: %s\n"), lpAppArray[dwIndex].szApiKey);
    _tprintf(_T("LUID: %s\n"), lpAppArray[dwIndex].szLuid);
    _tprintf(_T("Tokens: %lu\n"), lpAppArray[dwIndex].dwTokens);

    if (dwAppCount > 1 && dwIndex < dwAppCount - 1)
        _tprintf(_T("\n"));
}

LocalFree((HLOCAL)lpAppArray);
}

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebGetFirstApplication](#), [WebGetNextApplication](#), [WebRegisterAppId](#), [WebUnregisterAppId](#)

WebEnumObjects Function

```
BOOL WINAPI WebEnumObjects(  
    HSTORAGE hStorage,  
    LPCTSTR lpszMatchLabel,  
    LPCTSTR lpszContentType,  
    DWORD dwReserved,  
    LPWEB_STORAGE_OBJECT lpObjects,  
    LPDWORD lpdwObjects  
);
```

The **WebEnumObjects** function enumerates all storage objects that match the specified label or content type.

Parameters

hStorage

A handle to the storage container.

lpszMatchLabel

A pointer to a null terminated string which specifies the value to match against the object labels in the container. The string may contain wildcard characters similar to those use with the Windows filesystem. A "?" character matches any single character, and "*" matches any number of characters in the label. If this value is a NULL pointer or an empty string, all objects in the container will be matched.

lpszContentType

A pointer to a null terminated string which specifies the content type of the objects to be enumerated. If this value is a NULL pointer or an empty string, the content type is ignored and all matching objects are returned.

dwReserved

An unsigned integer value that is reserved for future use. This value must be zero.

lpObjects

A pointer to an array of [WEB_STORAGE_OBJECT](#) structures that will contain information about the enumerated objects. If this parameter is NULL, then no object information is returned.

lpdwObjects

A pointer to an unsigned integer that will contain the number of objects enumerated by this function. If the *lpObjects* parameter points to an array of [WEB_STORAGE_OBJECT](#) structures, this parameter must be initialized to the maximum size of the array being passed to the function. If the *lpObjects* parameter is NULL, this value must be initialized to zero, and when the function returns it will contain the number of matching objects.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebEnumObjects** function can be used to enumerate the number of objects that match a given label, content type or both. If a content type is specified, it must be a valid MIME media content type designated using the type/subtype nomenclature. For example, "text/plain" or "image/jpeg". An invalid MIME type will cause the function to fail.

This function can be used in two ways. If the *lpObjects* parameter is NULL and the value passed by reference in the *lpdwObjects* parameter is initialized to zero, the function will return the number of matching objects in *lpdwObjects*. This can be used to dynamically determine the size of the *lpObjects* array instead of declaring a fixed-size array in your application.

If the *lpObjects* parameter is not NULL, then the value referenced by *lpdwObjects* must be initialized to the maximum size of the array that *lpObjects* points to. It is important to note that if either parameter is not initialized correctly, it can result in memory corruption and/or an unhandled exception.

The **WebGetFirstObject** and **WebGetNextObject** functions can be used to iterate through all matching storage objects without requiring you to preallocate memory for an array. Using **WebGetFirstObject** and **WebGetNextObject** is more efficient when the container contains a large number of objects that match the specified label and/or content type.

Example

```
DWORD dwObjects = 0;

if (WebEnumObjects(hStorage, _T("*.pdf"), NULL, 0, NULL, &dwObjects))
{
    LPWEB_STORAGE_OBJECT lpObjects = NULL;
    DWORD dwIndex;

    // Return if there are no matching objects in the container
    if (dwObjects == 0)
        return;

    // Allocate memory for an array of WEB_STORAGE_OBJECT structures
    lpObjects = (LPWEB_STORAGE_OBJECT)LocalAlloc(LPTR,
sizeof(WEB_STORAGE_OBJECT) * dwObjects);

    if (lpObjects == NULL)
        return; // Exhausted virtual memory?

    // Enumerate the matching objects
    if (!WebEnumObjects(hStorage, NULL, NULL, 0, lpObjects, &dwObjects))
        return;

    // Print information about each object
    for (dwIndex = 0; dwIndex < dwObjects; dwIndex++)
    {
        _tprintf(_T("Object: %s\n"), lpObjects[dwIndex].szObjectId);
        _tprintf(_T("Label: %s\n"), lpObjects[dwIndex].szLabel);
        _tprintf(_T("Size: %lu\n"), lpObjects[dwIndex].dwObjectSize);
        _tprintf(_T("Digest: %s\n"), lpObjects[dwIndex].szDigest);
        _tprintf(_T("Content: %s\n"), lpObjects[dwIndex].szContent);

        if (dwObjects > 1 && dwIndex < dwObjects - 1)
            _tprintf(_T("\n"));
    }

    LocalFree((HLOCAL)lpObjects);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebGetFirstObject](#), [WebGetNextObject](#), [WebGetObjectInformation](#)

WebEventProc Function

```
VOID CALLBACK WebEventProc(  
    HSTORAGE hStorage,  
    UINT nEventId,  
    DWORD dwError,  
    DWORD_PTR dwParam  
);
```

The **WebEventProc** function is an application-defined callback function that processes events generated by the client.

Parameters

hStorage

A handle to the storage container.

nEventId

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **WebRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **WebRegisterEvent** function. The **WebEventProc** function is a placeholder for the application-defined function name.

To obtain information about the current status of the read or write operation, call the **WebGetTransferStatus** function. An event handler can cancel the current operation by calling the **WebCancelTransfer** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebCancelTransfer](#), [WebGetTransferStatus](#), [WebRegisterEvent](#)

WebGetAccountId Function

```
BOOL WINAPI WebGetAccountId(  
    LPCTSTR lpszAccountId,  
    INT nMaxLength  
);
```

The **WebGetAccountId** function returns the web services account ID associated with the current session.

Parameters

lpszAccountId

A pointer to a null-terminated string that will contain the account ID when the function returns. This parameter cannot be NULL and must be at least 35 characters in length.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the *lpszAccountId* string parameter, including the terminating null character.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The account ID is a string that uniquely identifies the web services account that is associated with the session. The account ID corresponds with your product serial number and runtime license key, but it is not identical to either of those values.



If you are using an evaluation license, the account ID is temporary and only valid during the evaluation period. After the evaluation period has expired, the account ID is revoked and objects stored using this ID will be deleted. It is not recommended that you store critical application data using an evaluation license.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebGetStorageId](#)

WebGetErrorString Function

```
INT WINAPI WebGetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);
```

The **WebGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **WebGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cswebv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebGetLastError](#), [WebSetLastError](#)

WebGetFile Function

```
BOOL WINAPI WebGetFile(  
    HSTORAGE hStorage,  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszObjectLabel  
);
```

The **WebGetFile** function downloads the contents of a storage object and copies it to a local file.

Parameters

hStorage

A handle to the storage container.

lpszLocalFile

A pointer to a null terminated string that specifies the name of the local file that will be created or overwritten with the contents of the storage object. If a path is not specified, the file will be created in the current working directory.

lpszObjectLabel

A pointer to a null terminated string that specifies the label of the object that should be retrieved from the server.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetFile** function downloads the contents of the storage object and stores it in a local file. For information about the stored object that has been downloaded, use the **WebGetFileEx** function.

If you are downloading a large object and want your application to receive progress updates during the data transfer, use the **WebRegisterEvent** function and provide a pointer to a callback function that will receive event notifications.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebDownloadFile](#), [WebGetFileEx](#), [WebGetObject](#), [WebPutFile](#), [WebPutObject](#), [WebRegisterEvent](#), [WebUploadFile](#)

WebGetFileEx Function

```
BOOL WINAPI WebGetFileEx(  
    HSTORAGE hStorage,  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszObjectLabel,  
    DWORD dwReserved,  
    LPWEB_STORAGE_OBJECT lpObject  
);
```

The **WebGetFileEx** function downloads the contents of a storage object and copies it to a local file.

Parameters

hStorage

A handle to the storage container.

lpszLocalFile

A pointer to a null terminated string that specifies the name of the local file that will be created or overwritten with the contents of the storage object. If a path is not specified, the file will be created in the current working directory.

lpszObjectLabel

A pointer to a null terminated string that specifies the label of the object that should be retrieved from the server.

dwReserved

An unsigned integer value that is reserved for future use. This value should always be zero.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain additional information about the object. This parameter may be NULL if the information is not required.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetFileEx** function downloads the contents of the storage object and stores it in a local file. Additional metadata about the object will be returned in the [WEB_STORAGE_OBJECT](#) structure provided by the caller, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents.

If you are downloading a large object and want your application to receive progress updates during the data transfer, use the **WebRegisterEvent** function and provide a pointer to a callback function that will receive event notifications.

The **WebDownloadFile** and **WebDownloadFileEx** functions provide similar functionality, but do not require a handle to an open storage container.

Example

```
WEB_STORAGE_OBJECT webObject;  
  
// Download the object and store the contents in a local file  
if (WebGetFileEx(hStorage, lpszLocalFile, lpszObjectLabel, 0, &webObject))
```

```

{
    // The object was downloaded, display the metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %I64u\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The object could not be retrieved, display the error
    TCHAR szError[128];

    WebGetErrorString(WebGetLastError(), szError, 128);
    _tprintf(_T("Unable to retrieve \"%s\" (%s)\n"), lpszObjectLabel, szError);
}

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebDownloadFile](#), [WebGetFile](#), [WebGetObjectEx](#), [WebPutFileEx](#), [WebPutObjectEx](#),
[WebRegisterEvent](#), [WebUploadFile](#)

WebGetLastError Function

```
DWORD WINAPI WebGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **WebSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **WebGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **WebSetLastError(0)** when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or WEB_ERROR. Those functions which call **WebSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswebv10.lib

See Also

[WebGetErrorString](#), [WebSetLastError](#)

WebGetFirstApplication Function

```
BOOL WINAPI WebGetFirstApplication(  
    LPWEB_STORAGE_APPLICATION LpAppInfo,  
    LPDWORD LpdwContext,  
);
```

The **WebGetFirstApplication** function returns information about the first registered application for the current storage account.

Parameters

lpAppInfo

A pointer to a [WEB_STORAGE_APPLICATION](#) structure that will contain information about the registered application when the function returns. This parameter cannot be NULL.

lpdwContext

A pointer to an unsigned integer that is used with subsequent calls to the **WebGetNextApplication** function. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetFirstApplication** function returns information about the first registered application. It is used in conjunction with the **WebGetNextApplication** function to enumerate all of the registered application IDs associated with your storage account.

This function provides an alternative to the **WebEnumApplications** function, which populates an array of [WEB_STORAGE_APPLICATION](#) structures. In some cases, iterating through each AppID in a loop may be preferred to preallocating memory for an array to store every registered application. Using **WebGetFirstApplication** and **WebGetNextApplication** is more efficient when you have registered a large number of application IDs.

You cannot intermix calls between **WebGetFirstApplication** and **WebEnumApplications**. The **WebEnumApplications** function will reset the internal application ID cache for the client session and subsequent calls to **WebGetNextApplication** will fail. The cached application ID information used by this function is shared by the entire process. Attempting to enumerate all registered application IDs in multiple threads at the same time can yield unexpected results. It is recommended that multi-threaded clients use a critical section to ensure that only a single thread is enumerating the AppIDs at any one time.

Example

```
WEB_STORAGE_APPLICATION webApp;  
DWORD dwContext = 0;  
  
if (WebGetFirstApplication(&webApp, &dwContext))  
{  
    do  
    {  
        // Print information for each registered application  
        _tprintf(_T("AppId: %s\n"), webApp.szAppId);  
        _tprintf(_T("Key: %s\n"), webApp.szApiKey);  
        _tprintf(_T("LUID: %s\n"), webApp.szLuid);  
    }  
}
```

```
        _tprintf(_T("Tokens: %lu\n"), webApp.dwTokens);
        _tprintf(_T("\n"));
    }
    while (WebGetNextApplication(&webApp, &dwContext));
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebEnumApplications](#), [WebGetNextApplication](#), [WebRegisterAppId](#), [WebUnregisterAppId](#)

WebGetFirstObject Function

```
BOOL WINAPI WebGetFirstObject(  
    HSTORAGE hStorage,  
    LPCTSTR lpszMatchLabel,  
    LPCTSTR lpszContentType,  
    DWORD dwReserved,  
    LPWEB_STORAGE_OBJECT lpObject  
);
```

The **WebGetFirstObject** function returns information about the first storage object that matches the specified label or content type.

Parameters

hStorage

A handle to the storage container.

lpszMatchLabel

A pointer to a null terminated string which specifies the value to match against the object labels in the container. The string may contain wildcard characters similar to those use with the Windows filesystem. A "?" character matches any single character, and "*" matches any number of characters in the label. If this value is a NULL pointer or an empty string, all objects in the container will be matched.

lpszContentType

A pointer to a null terminated string which specifies the content type of the objects to be enumerated. If this value is a NULL pointer or an empty string, the content type is ignored and all matching objects are returned. If a content type is specified, it must be a valid MIME media content type designated using the **type/subtype** nomenclature.

dwReserved

An unsigned integer value that is reserved for future use. This value must be zero.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain information about the storage object when the function returns. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetFirstObject** function returns information about the first object that matches a given label, content type or both. It is used in conjunction with the **WebGetNextObject** function to enumerate all of the matching objects in the storage container.

This function provides an alternative to the **WebEnumObjects** function, which populates an array of [WEB_STORAGE_OBJECT](#) structures. In some cases, iterating over each object in a loop may be preferred to preallocating memory for an array to store every matching object. Using **WebGetFirstObject** and **WebGetNextObject** is more efficient when the container contains a large number of objects that match the specified label and/or content type.

You cannot intermix calls between **WebGetFirstObject** and **WebEnumObjects**. The **WebEnumObjects** function will reset the internal object cache for the client session and

subsequent calls to **WebGetNextObject** will fail.

Example

```
WEB_STORAGE_OBJECT webObject;

if (WebGetFirstObject(hStorage, _T("*.pdf"), NULL, 0, &webObject))
{
    do
    {
        // Print information about each object
        _tprintf(_T("Object: %s\n"), webObject.szObjectId);
        _tprintf(_T("Label: %s\n"), webObject.szLabel);
        _tprintf(_T("Size: %lu\n"), webObject.dwObjectSize);
        _tprintf(_T("Digest: %s\n"), webObject.szDigest);
        _tprintf(_T("Content: %s\n"), webObject.szContent);
        _tprintf(_T("\n"));
    }
    while (WebGetNextObject(hStorage, &webObject));
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebEnumObjects](#), [WebGetNextObject](#)

WebGetLocation Function

```
BOOL WINAPI WebGetLocation(  
    LPWEB_LOCATION lpLocation  
);
```

The **WebGetLocation** function returns the current physical location of the local computer system.

Parameters

lpLocation

A pointer to a [WEB_LOCATION](#) structure that will contain information about the location when the function returns. The structure members will be automatically initialized to zero and zero-length strings when the function is called. This parameter cannot be a NULL pointer.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetLocation** function is used to obtain physical location information associated with the with the external IP address of the local system. The accuracy of this information can vary depending on the location, with the most detailed information being available for North America. The country and time zone information for all locations is generally accurate. However, as the location information becomes more precise, details such as city names, postal codes and specific geographic locations (e.g.: longitude and latitude) may have reduced accuracy.

This location information should not be used by programs that require extremely accurate map coordinates, such as navigation applications. The location information in North America should be generally accurate within a 25 mile (40km) radius. However, given the nature of how IP address location works, there is no guarantee that location information for any specific IP address or network will be accurate.

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

Example

```
WEB_LOCATION webLocation;  
  
// Return information about the location of the local system  
if (WebGetLocation(&webLocation))  
{  
    _tprintf(_T("Location ID: %s\n"), webLocation.szLocationId);  
    _tprintf(_T("IP address: %s\n"), webLocation.szIPAddress);  
    _tprintf(_T("ASN: %u\n"), webLocation.nAutonomousSystemNumber);  
    _tprintf(_T("Organization: %s\n"), webLocation.szOrganization);  
    _tprintf(_T("Region Name: %s\n"), webLocation.szRegionName);  
    _tprintf(_T("Region Code: %03u\n"), webLocation.nRegionCode);  
    _tprintf(_T("Country Name: %s\n"), webLocation.szCountryName);  
    _tprintf(_T("Country Code: %03u\n"), webLocation.nCountryCode);  
    _tprintf(_T("Subdivision: %s\n"), webLocation.szSubdivision);  
}
```

```

    _tprintf(_T("City Name: %s\n"), webLocation.szCityName);
    _tprintf(_T("Postal Code: %s\n"), webLocation.szPostalCode);
    _tprintf(_T("Timezone: %s (%ld)\n"), webLocation.szTimezone,
webLocation.nTimezoneOffset);
    _tprintf(_T("Local Time: %04d-%02d-%02d %02d:%02d:%02d %s\n"),
        webLocation.stLocalTime.wYear,
        webLocation.stLocalTime.wMonth,
        webLocation.stLocalTime.wDay,
        webLocation.stLocalTime.wHour,
        webLocation.stLocalTime.wMinute,
        webLocation.stLocalTime.wSecond,
        webLocation.szTzShortName);
    _tprintf(_T("Coordinates: %s\n"), webLocation.szCoordinates);
    _tprintf(_T("Latitude: %.4f\n"), webLocation.dLatitude);
    _tprintf(_T("Longitude: %.4f\n"), webLocation.dLongitude);
}
else
{
    _tprintf(_T("Unable to determine the current location\n"));
}

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

WebGetNextApplication Function

```
BOOL WINAPI WebGetNextApplication(  
    LPWEB_STORAGE_APPLICATION LpAppInfo,  
    LPDWORD LpdwContext,  
);
```

The **WebGetNextApplication** function returns information about the next registered application for the current storage account.

Parameters

lpAppInfo

A pointer to a [WEB_STORAGE_APPLICATION](#) structure that will contain information about the registered application when the function returns. This parameter cannot be NULL.

lpdwContext

A pointer to an unsigned integer that is used with subsequent calls to the **WebGetNextApplication** function. The value of this parameter is initialized by calling the **WebGetFirstApplication** function. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetNextApplication** function returns information about the next registered application after initially calling the **WebGetFirstApplication** function. It is used to enumerate all of the registered application IDs associated with your storage account.

This function provides an alternative to the **WebEnumApplications** function, which populates an array of [WEB_STORAGE_APPLICATION](#) structures. In some cases, iterating through each AppID in a loop may be preferred to preallocating memory for an array to store every registered application. Using **WebGetFirstApplication** and **WebGetNextApplication** is more efficient when you have registered a large number of application IDs.

You cannot intermix calls between **WebGetNextApplication** and **WebEnumApplications**. The **WebEnumApplications** function will reset the internal application ID cache for the client session and subsequent calls to **WebGetNextApplication** will fail. The cached application ID information used by this function is shared by the entire process. Attempting to enumerate all registered application IDs in multiple threads at the same time can yield unexpected results. It is recommended that multi-threaded clients use a critical section to ensure that only a single thread is enumerating the AppIDs at any one time.

Example

```
WEB_STORAGE_APPLICATION webApp;  
DWORD dwContext = 0;  
  
if (WebGetFirstApplication(&webApp, &dwContext))  
{  
    do  
    {  
        // Print information for each registered application  
        _tprintf(_T("AppId: %s\n"), webApp.szAppId);  
        _tprintf(_T("Key: %s\n"), webApp.szApiKey);  
    }  
}
```

```
        _tprintf(_T("LUID: %s\n"), webApp.szLuid);
        _tprintf(_T("Tokens: %lu\n"), webApp.dwTokens);
        _tprintf(_T("\n"));
    }
    while (WebGetNextApplication(&webApp, &dwContext));
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebEnumApplications](#), [WebGetFirstApplication](#), [WebRegisterAppId](#), [WebUnregisterAppId](#)

WebGetNextObject Function

```
BOOL WINAPI WebGetNextObject(  
    HSTORAGE hStorage,  
    LPWEB_STORAGE_OBJECT lpObject  
);
```

The **WebGetNextObject** function returns information about the next storage object that matches the specified label or content type.

Parameters

hStorage

A handle to the storage container.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain information about the storage object when the function returns. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetNextObject** function returns information about the next object that matches the label and/or content type that was specified by the **WebGetFirstObject** function call. This function may only be called after **WebGetFirstObject** has been called, otherwise it will fail.

When all matching objects have been returned to the caller, this function will return zero (FALSE) and the **WebGetLastError** function will return NO_ERROR. Any other error code indicates an underlying problem with the request, such as an invalid parameter passed to the function.

You cannot intermix calls between **WebGetNextObject** and **WebEnumObjects**. The **WebEnumObjects** function will reset the internal object cache for the client session and subsequent calls to **WebGetNextObject** will fail.

Example

```
WEB_STORAGE_OBJECT webObject;  
  
if (WebGetFirstObject(hStorage, _T("*.pdf"), NULL, 0, &webObject))  
{  
    do  
    {  
        // Print information about each object  
        _tprintf(_T("Object: %s\n"), webObject.szObjectId);  
        _tprintf(_T("Label: %s\n"), webObject.szLabel);  
        _tprintf(_T("Size: %lu\n"), webObject.dwObjectSize);  
        _tprintf(_T("Digest: %s\n"), webObject.szDigest);  
        _tprintf(_T("Content: %s\n"), webObject.szContent);  
        _tprintf(_T("\n"));  
    }  
    while (WebGetNextObject(hStorage, &webObject));  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebEnumObjects](#), [WebGetFirstObject](#)

WebGetObject Function

```
BOOL WINAPI WebGetObject(  
    HSTORAGE hStorage,  
    LPCTSTR lpszObjectLabel,  
    LPVOID lpvBuffer,  
    DWORD lpdwLength  
);
```

The **WebGetObject** function retrieves the contents of a storage object and copies it to the memory buffer that is provided.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A pointer to a null terminated string that specifies the label of the object that should be retrieved from the server.

lpvBuffer

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvBuffer* parameter. If the *lpvBuffer* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the file that was downloaded.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetObject** function is used to retrieve a stored object from the server and copy it into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the object. In this case, the *lpvBuffer* parameter will point to the buffer that was allocated and the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvBuffer* parameter point to a global memory handle which will contain the file data when the function returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

Example

```
HGLOBAL hgb1Buffer = (HGLOBAL)NULL;  
DWORD dwLength = 0;
```

```
// Return the file data into block of global memory allocated by  
// the GlobalAlloc function; the handle to this memory will be
```

```
// returned in the hglobalBuffer parameter
if (WebGetObject(hStorage, lpzObjectLabel, &hglobalBuffer, &dwLength))
{
    // Lock the global memory handle, returning a pointer to the
    // resource data
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hglobalBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hglobalBuffer);
    GlobalFree(hglobalBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebGetFile](#), [WebGetObjectEx](#), [WebPutFile](#), [WebPutObject](#)

WebGetObjectEx Function

```
BOOL WINAPI WebGetObjectEx(  
    HSTORAGE hStorage,  
    LPCTSTR lpszObjectLabel,  
    DWORD dwReserved,  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    LPWEB_STORAGE_OBJECT lpObject,  
);
```

The **WebGetObjectEx** function retrieves the contents of a storage object and copies it to the memory buffer that is provided. Additional information about the object is returned to the caller.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A pointer to a null terminated string that specifies the label of the object that should be retrieved from the server.

dwReserved

An unsigned integer that is reserved for future use. This value should always be zero.

lpvBuffer

A pointer to a byte buffer which will contain the data transferred from the server, or a pointer to a global memory handle which will reference the data when the function returns.

lpdwLength

A pointer to an unsigned integer which should be initialized to the maximum number of bytes that can be copied to the buffer specified by the *lpvBuffer* parameter. If the *lpvBuffer* parameter points to a global memory handle, the length value should be initialized to zero. When the function returns, this value will be updated with the actual length of the file that was downloaded.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain additional information about the object. This parameter may be NULL if the information is not required.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetObjectEx** function is used to retrieve a stored object from the server and copy it into a local buffer. The function may be used in one of two ways, depending on the needs of the application. The first method is to pre-allocate a buffer large enough to store the contents of the object. In this case, the *lpvBuffer* parameter will point to the buffer that was allocated and the value that the *lpdwLength* parameter points to should be initialized to the size of that buffer.

The second method that can be used is have the *lpvBuffer* parameter point to a global memory handle which will contain the file data when the function returns. In this case, the value that the *lpdwLength* parameter points to must be initialized to zero. It is important to note that the

memory handle returned by the function must be freed by the application, otherwise a memory leak will occur. See the example code below.

If you wish to retrieve the contents of a text object and store it in a null terminated string buffer, use the **WebGetTextObject** function. If you wish to download the contents of a stored file, use the **WebGetFileEx** function.

Additional metadata about the object will be returned in the WEB_STORAGE_OBJECT structure provided by the caller, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents.

Example

```
HGLOBAL hgblBuffer = (HGLOBAL)NULL;
DWORD dwLength = 0;
WEB_STORAGE_OBJECT webObject;

// Return the file data into block of global memory allocated by
// the GlobalAlloc function; the handle to this memory will be
// returned in the hgblBuffer parameter
if (WebGetObjectEx(hStorage, lpszObjectLabel, 0, &hgblBuffer, &dwLength,
&webObject))
{
    // Lock the global memory handle, returning a pointer to the
    // resource data
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // After the data has been used, the handle must be unlocked
    // and freed, otherwise a memory leak will occur
    GlobalUnlock(hgblBuffer);
    GlobalFree(hgblBuffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebGetFileEx](#), [WebGetObject](#), [WebGetTextObject](#), [WebPutFileEx](#), [WebPutObjectEx](#)

WebGetObjectInformation Function

```
BOOL WINAPI WebGetObjectInformation(  
    HSTORAGE hStorage,  
    LPCTSTR lpszObjectLabel,  
    DWORD dwReserved,  
    LPWEB_STORAGE_OBJECT lpObject  
);
```

The **WebGetObjectInformation** function retrieves the metadata for a storage object.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A pointer to a null terminated string that specifies the label of the storage object.

dwReserved

An unsigned integer that is reserved for future use. This value should always be zero.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain additional information about the object. This parameter cannot be a NULL pointer.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetObjectInformation** function is used to retrieve the metadata for a stored object on the server, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents. This function can also be used as a simple method to determine if the specified object exists without the overhead of requesting the server attempt to retrieve the contents of an object.

Although storage object labels are similar to Windows file names, they are case-sensitive. When requesting information about an object, your application must specify the label name exactly as it was created. The object label cannot contain wildcard characters.

If you are only interested in obtaining the size of a stored object, you can use the **WebGetObjectSize** function.

To obtain information about how much storage your applications are using and the total number of stored objects, use the **WebGetStorageQuota** function.

Example

```
WEB_STORAGE_OBJECT webObject;  
  
// Get information about the object  
if (WebGetObjectInformation(hStorage, lpszObjectLabel, 0, &webObject))  
{  
    // The object exists, display the metadata  
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);  
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);  
}
```

```
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The object does not exist, display the error
    TCHAR szError[128];

    WebGetErrorString(WebGetLastError(), szError, 128);
    _tprintf(_T("Unable to get information on \"%s\" (%s)\n"), lpszObjectLabel,
szError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebGetFile](#), [WebGetObject](#), [WebGetObjectSize](#), [WebGetStorageQuota](#), [WebPutFile](#),
[WebPutObject](#)

WebGetObjectSize Function

```
BOOL WINAPI WebGetObjectSize(  
    HSTORAGE hStorage,  
    LPCTSTR lpszObjectLabel,  
    LPDWORD lpdwObjectSize  
);
```

The **WebGetObjectSize** function returns the size of the stored object.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A pointer to a null terminated string that specifies the label of the storage object. This parameter cannot be a NULL pointer.

lpdwObjectSize

A pointer to an unsigned integer value that will contain the size of the object. If this parameter is NULL, the parameter is ignored and the function only checks for the existence of an object that matches the specified label.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetObjectSize** function is used to retrieve the size of a stored object on the server. This function can also be used as a simple method to determine if the specified object exists. An object size of zero means the object exists, but currently has no data associated with it.

If the *lpdwObjectSize* parameter is not NULL, its value is initialized to zero when the function is called, and updated with the object size when the function returns.

Although storage object labels are similar to Windows file names, they are case-sensitive. When requesting the size of an object, your application must specify the label name exactly as it was created. The object label cannot contain wildcard characters.

The **WebGetObjectInformation** function can be used to obtain the metadata associated with the storage object, including the size, content type, creation date and a SHA-256 digest of the data.

To obtain information about how much storage your applications are using and the total number of stored objects, use the **WebGetStorageQuota** function.

Example

```
// Check if the object exists  
DWORD dwObjectSize;  
  
if (WebGetObjectSize(hStorage, lpszObjectLabel, &dwObjectSize))  
{  
    // The object exists, display the size in bytes  
    _tprintf(_T("The size of \"%s\" is %lu bytes\n"), lpszObjectLabel,  
dwObjectSize);  
}
```

```
else
{
    // The object size could not be determined, display the error
    TCHAR szError[128];

    WebGetErrorString(WebGetLastError(), szError, 128);
    _tprintf(_T("Unable to get the size of \"%s\" (%s)\n"), lpszObjectLabel,
szError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebGetObject](#), [WebGetObjectInformation](#), [WebGetStorageQuota](#), [WebPutObject](#)

WebGetStorageId Function

```
BOOL WINAPI WebGetStorageId(  
    HSTORAGE hStorage,  
    LPCTSTR lpszStorageId,  
    INT nMaxLength  
);
```

The **WebGetStorageId** function returns the storage container ID.

Parameters

hStorage

A handle to the storage container.

lpszStorageId

A pointer to a null-terminated string that will contain the storage ID when the function returns. This parameter cannot be NULL and must be at least 35 characters in length.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the *lpszStorageId* string parameter, including the terminating null character.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The storage ID is a string that identifies the storage container that was opened with the **WebOpenStorage** function. The storage ID is associated with your development license and is guaranteed to be a unique value.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebGetAccountId](#), [WebOpenStorage](#)

WebGetStorageQuota Function

```
BOOL WINAPI WebGetStorageQuota(  
    LPWEB_STORAGE_QUOTA lpQuota  
);
```

The **WebGetStorageQuota** function returns quota limits assigned to your storage account.

Parameters

lpQuota

A pointer to a [WEB_STORAGE_QUOTA](#) structure that will contain information the quota limits for your account when the function returns. This parameter cannot be NULL.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetStorageQuota** function can be used to determine how much storage space is available to your application. The *ulBytesFree* structure member will tell you how many bytes of storage you have available, and the *dwObjectSize* member will tell you the maximum size of any individual storage object. In addition to the bytes allocated for storage, there is also a limit on the total number of objects that your application may create which is specified by the *dwObjectLimit* member.

Accounts that are created with an evaluation license have much lower quota limits than a standard account and should be used for testing purposes only. After the evaluation period has ended, all objects stored using the evaluation license will be deleted.

Example

```
WEB_STORAGE_QUOTA appQuota;  
  
// Initialize the library  
if (WebInitialize(CSTOOLS10_LICENSE_KEY, NULL) == FALSE)  
{  
    _tprintf(_T("Unable to initialize the SocketTools library\n"));  
    return;  
}  
  
// Display storage account usage and limits  
if (WebGetStorageQuota(&appQuota))  
{  
    _tprintf(_T("Objects Used:   %lu\n"), appQuota.dwObjects);  
    _tprintf(_T("Object Limit:  %lu\n"), appQuota.dwObjectLimit);  
    _tprintf(_T("Object Size:   %lu\n"), appQuota.dwObjectSize);  
    _tprintf(_T("Bytes Used:    %I64u\n"), appQuota.ulBytesUsed);  
    _tprintf(_T("Bytes Free:    %I64u\n"), appQuota.ulBytesFree);  
    _tprintf(_T("Storage Limit: %I64u\n"), appQuota.ulStorageLimit);  
}  
else  
{  
    // Unable to get the quota information for this account  
    TCHAR szError[128];  
  
    WebGetErrorString(WebGetLastError(), szError, 128);
```

```
    _tprintf(_T("Unable to get the account quota (%s)\n"), szError);  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebGetObjectInformation](#), [WebGetObjectSize](#), [WEB_STORAGE_QUOTA](#)

WebGetStorageTimeout Function

```
DWORD WINAPI WebGetStorageTimeout(  
    HSTORAGE hStorage  
);
```

The **WebGetStorageTimeout** function returns the number of seconds the client will wait for a response from the storage server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The default timeout period is 10 seconds.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

See Also

[WebOpenStorage](#), [WebSetStorageTimeout](#)

WebGetTextObject Function

```
BOOL WINAPI WebGetTextObject(  
    HSTORAGE hStorage,  
    LPCTSTR lpszObjectLabel,  
    LPTSTR lpszBuffer,  
    INT nMaxLength,  
    LPWEB_STORAGE_OBJECT lpObject,  
);
```

The **WebGetTextObject** function retrieves the contents of a text storage object and copies it to the string buffer that is provided. Additional information about the object is returned to the caller.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A pointer to a null terminated string that specifies the label of the text object that should be retrieved from the server.

lpszBuffer

A pointer to a string buffer which will contain the text stored in the object. The string must be large enough to store the terminating null character.

nMaxLength

An integer value that specifies the maximum number of characters that can be copied into the string buffer. This value must be large enough to store the complete text and the terminating null character.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain additional information about the object. This parameter may be NULL if the information is not required.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetTextObject** function is used to retrieve the text from a stored object and copy it into a string buffer. The string buffer must be pre-allocated and large enough to contain the entire text and a terminating null character.

If you use the **WebGetObjectSize** function to determine the size of the object so your application can dynamically allocate memory for the string buffer, be sure to allocate a string buffer that is larger than the reported size. The size of the stored text object may differ slightly from what is returned in the string, depending on how the text has been encoded.

Text objects that are created or updated using the Unicode version of **WebPutTextObject** function will always be converted and stored as UTF-8 encoded text. If your application uses the ANSI version of this function to retrieve that text, it will be UTF-8 encoded and your application will need to convert it back to UTF-16 encoded text, if needed. The Unicode version of this function will perform the conversion automatically.

This function should never be used to retrieve data from an object that does not contain text. If

the **WebPutObject** function was previously used to store UTF-16 encoded text, you must use **WebGetObject** to retrieve that data. This function only recognizes UTF-8 encoded text and considers UTF-16 encoded text to be binary data.

Additional metadata about the object will be returned in the [WEB_STORAGE_OBJECT](#) structure provided by the caller, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents.

Example

```
LPTSTR lpszBuffer = (LPTSTR)NULL;
INT nMaxLength = 8192;
WEB_STORAGE_OBJECT webObject;

lpszBuffer = (LPTSTR)LocalAlloc(LPTR, nMaxLength);
if (lpszBuffer == NULL)
{
    // Unable to allocate memory for the string
    _ftprintf(stderr, _T("Unable to allocate %d bytes of memory\n"),
nMaxLength);
    return;
}

if (WebGetTextObject(hStorage, lpszObjectLabel, lpszBuffer, nMaxLength,
&webObject))
{
    // Output the text contained in the object
    _fputts(lpszBuffer, stdout);
}
else
{
    DWORD dwError = WebGetLastError();
    TCHAR szError[128];

    WebGetErrorString(dwError, szError, 128);
    _ftprintf(stderr, _T("Unable to get \"%s\": %s\n"), lpszObjectLabel,
szError);
}

// Free memory allocated for the text when no longer needed
LocalFree((HLOCAL)lpszBuffer);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebCompareText](#), [WebGetFileEx](#), [WebGetObject](#), [WebPutFileEx](#), [WebPutObjectEx](#),
[WebPutTextObject](#)

WebGetTransferStatus Function

```
BOOL WINAPI WebGetTransferStatus(  
    HSTORAGE hStorage,  
    LPWEB_STORAGE_TRANSFER lpStatus  
);
```

The **WebGetTransferStatus** function returns information about the current data transfer in progress.

Parameters

hStorage

Handle to the storage container.

lpStatus

A pointer to a [WEB_STORAGE_TRANSFER](#) structure which contains information about the status of the current data transfer.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebGetTransferStatus** function returns information about the current data transfer, including the average number of bytes transferred per second and the estimated amount of time until the transfer completes. If there is no data currently being transferred, this function will return the status of the last successful data transfer.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified storage session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cswebv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebCancelTransfer](#), [WebRegisterEvent](#)

WebInitialize Function

```
BOOL WINAPI WebInitialize(  
    LPCTSTR lpszLicenseKey,  
    LPINITDATA lpData  
);
```

The **WebInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpData

Pointer to an INITDATA data structure. This parameter may be NULL if the initialization data for the library is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must insure that this function is called with a valid license key.

If the *lpData* argument is specified, it must point to an INITDATA structure which has been initialized by setting the *dwSize* member to the size of the structure. All other structure members should be set to zero. If the function is successful, then the INITDATA structure will be filled with identifying information about the library.

Although it is only required that **WebInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **WebUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **WebInitialize** or **WebUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

WebMoveObject Function

```
BOOL WINAPI WebMoveObject(  
    HSTORAGE hStorage,  
    LPCTSTR lpszOldLabel,  
    LPCTSTR lpszNewLabel,  
    DWORD dwStorageType,  
    LPWEB_STORAGE_OBJECT lpObject  
);
```

The **WebMoveObject** function moves an existing object to a different storage container.

Parameters

hStorage

A handle to the storage container.

lpszOldLabel

A pointer to a null terminated string which specifies the name of the existing storage object to be moved. This parameter must specify a valid object label and cannot be a NULL pointer or an empty string.

lpszNewLabel

A pointer to a null terminated string which specifies a new label for the storage object being moved. This parameter may be NULL or point to an empty string, in which case the label name is not changed.

dwStorageType

An integer value that identifies the storage container type. One of the following values should be specified:

Constant	Description
WEB_STORAGE_GLOBAL (1)	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN (2)	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
WEB_STORAGE_MACHINE (3)	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.

WEB_STORAGE_USER
(4)

Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain information about the new storage object. If this information is not required, this parameter may be a NULL pointer.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebMoveObject** function is used to move an existing storage object to a new container. For example, it can move an object originally created using WEB_STORAGE_USER to the WEB_STORAGE_MACHINE container. The *dwStorageType* parameter must specify a valid storage container type and cannot be WEB_STORAGE_DEFAULT.

If the *dwStorageType* parameter specifies the same container that the object is currently in, and the *lpszNewLabel* parameter specifies a new label name, this function will simply rename the existing object and is effectively the same as calling the **WebRenameObject** function.

To duplicate an existing storage object, use the **WebCopyObject** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebCopyObject](#), [WebDeleteObject](#), [WebRenameObject](#)

WebOpenStorage Function

```
HSTORAGE WINAPI WebOpenStorage(  
    LPCTSTR lpszAppId,  
    DWORD dwStorageType
```

The **WebOpenStorage** function establishes a connection with the server and opens the storage container associated with the specified storage type.

Parameters

lpszAppId

A pointer to null terminated string which specifies the application ID for the storage container. The application ID is a string that uniquely identifies the application and can only contain letters, numbers, the period and the underscore character. If this parameter is NULL or an empty string, the default identifier **SocketTools.Storage.Default** will be used.

dwStorageType

An integer value that identifies the storage container type. One of the following values should be specified:

Constant	Description
WEB_STORAGE_GLOBAL (1)	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN (2)	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
WEB_STORAGE_MACHINE (3)	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER (4)	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

Return Value

If the function succeeds, the return value is a handle to the storage container. If the function fails, the return value is `INVALID_HANDLE`. To get extended error information, call **WebGetLastError**.

Remarks

The **WebOpenStorage** function opens the specified storage container and requests an access token for the application. This is the first function that must be called prior to accessing any stored objects.

The application ID is a string that uniquely identifies the application requesting the access and must have been previously registered with the server by calling the **WebRegisterAppId** function. If the *lpszAppId* parameter is `NULL` or an empty string, the function will use a default internal ID that is allocated for each storage account. You can use this default ID if you wish to share data between all of the applications you create.

The storage type specifies the type of container that objects will be stored in. In most cases, we recommend using `WEB_STORAGE_GLOBAL` which means that stored objects will be accessible to all users of your application. However, you can limit access to the stored objects based on the local domain, local machine ID or the current user SID.

If you specify anything other than global storage, objects can be orphaned if the system configuration changes. For example, if `WEB_STORAGE_MACHINE` is specified, the objects that are stored there can only be accessed from that computer system. If the system is reconfigured (for example, the boot volume formatted and Windows is reinstalled) the unique identifier for that system will change and the previous objects that were stored by your application can no longer be accessed.

It is advisable to store critical application data and configuration information using `WEB_STORAGE_GLOBAL` and use other non-global storage containers for configuration information that is unique to that system and/or user which is not critical and can be easily recreated.

Example

```
HSTORAGE hStorage = INVALID_HANDLE;
LPCTSTR lpszAppId = _T("MyCompany.WebTest.1");
LPCTSTR lpszLocalFile = _T("TestDocument.pdf");
LPCTSTR lpszObjectLabel = _T("TestDocument.pdf");
WEB_STORAGE_OBJECT webObject = { 0, };
BOOL bSuccess = FALSE;

// Register the application ID which identifies the application
if (!WebRegisterAppId(lpszAppId))
{
    tprintf(_T("Unable to register the application ID"));
    _exit(0);
}

// Open the application storage container
hStorage = WebOpenStorage(lpszAppId, WEB_STORAGE_GLOBAL);

if (hStorage == INVALID_HANDLE)
{
    tprintf(_T("Unable to open global storage for this application"));
    _exit(0);
}
```

```

// Upload a local file to the storage server and return information
// about the stored object when it completes
bSuccess = WebPutFileEx(hStorage,
                        lpszLocalFile,
                        lpszObjectLabel,
                        NULL,
                        WEB_OBJECT_DEFAULT,
                        0,
                        &webObject);

if (bSuccess)
{
    // Print information about the object that was created
    _tprintf(_T("Object: %s\n"), webObject.szObjectId);
    _tprintf(_T("Label: %s\n"), webObject.szLabel);
    _tprintf(_T("Size: %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest: %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The upload failed, display error information
    DWORD dwError = WebGetLastError();
    TCHAR szError[128];

    WebGetErrorString(dwError, szError, 128);
    _tprintf(_T("WebPutFileEx failed, error 0x%08lX (%s)\n"), dwError, szError);
}

// Release the handle allocated for this storage session
WebCloseStorage(hStorage);

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebCloseStorage](#), [WebGetFile](#), [WebGetObject](#), [WebGetStorageId](#), [WebPutFile](#), [WebPutObject](#), [WebRegisterAppId](#)

WebPutFile Function

```
BOOL WINAPI WebPutFile(  
    HSTORAGE hStorage,  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszObjectLabel  
);
```

The **WebPutFile** function uploads the contents of a local file to a storage container.

Parameters

hStorage

A handle to the storage container.

lpszLocalFile

A pointer to a null terminated string that specifies the name of the local file that will be uploaded. If a path is not specified, the file will be read from the current working directory. The current user must have read access to the file, and an error will be returned if the function cannot obtain an exclusive lock on the file during the upload process.

lpszObjectLabel

A pointer to a null terminated string that specifies the label of the storage object that will be created or replaced. This parameter cannot be NULL or a zero-length string. The function will fail if the label contains any illegal characters.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebPutFile** function uploads the contents of a local file to the storage container specified by the *hStorage* handle. If an object with this name already exists, its data will be replaced with the contents of the file. The object will be created using the default attributes that permit read and write access, and will automatically determine the content type based on the type of file being uploaded.

If the label identifies an object that already exists in the container, and that object was created with the WEB_OBJECT_READONLY attribute, this function will fail. To replace a read-only object, the application must first move, rename or delete the existing object.

If you want to specify the object content type, its attributes, or obtain additional information about the storage object that was created, use the **WebPutFileEx** function.

The **WebValidateLabel** function can be used to ensure a label is valid prior to calling this function. Refer to that function for more information about the difference between Windows file names and object labels.

If you are uploading a large file and want your application to receive progress updates during the data transfer, use the **WebRegisterEvent** function and provide a pointer to a callback function that will receive event notifications.

Example

```
// Upload a local file to the storage container  
if (WebPutFile(hStorage, lpszLocalFile, lpszObjectLabel)
```

```
{
    // The file was uploaded successfully
    _tprintf(_T("The file was stored as \"%s\"\n"), lpszObjectLabel);
}
else
{
    // The file could not be uploaded, display the error
    TCHAR szError[128];

    WebGetErrorString(WebGetLastError(), szError, 128);
    _tprintf(_T("Unable to store \"%s\" (%s)\n"), lpszLocalFile, szError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebDownloadFile](#), [WebGetFile](#), [WebPutFileEx](#), [WebRegisterEvent](#), [WebUploadFile](#),
[WebValidateLabel](#)

WebPutFileEx Function

```
BOOL WINAPI WebPutFileEx(  
    HSTORAGE hStorage,  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszObjectLabel,  
    LPCTSTR lpszContentType,  
    DWORD dwAttributes,  
    DWORD dwReserved,  
    LPWEB_STORAGE_OBJECT LpObject  
);
```

The **WebPutFileEx** function uploads the contents of a local file to a storage container and returns information about the new object.

Parameters

hStorage

A handle to the storage container.

lpszLocalFile

A pointer to a null terminated string that specifies the name of the local file that will be uploaded. If a path is not specified, the file will be read from the current working directory. The current user must have read access to the file, and an error will be returned if the function cannot obtain an exclusive lock on the file during the upload process.

lpszObjectLabel

A pointer to a null terminated string that specifies the label of the storage object that will be created or replaced. This parameter cannot be NULL or a zero-length string. The function will fail if the label contains any illegal characters.

lpszContentType

A pointer to a null terminated string that identifies the contents of the file being uploaded. If this parameter is a NULL pointer, or specifies a zero-length string, the function will attempt to automatically determine the content type based on the file name extension and the contents of the file.

dwAttributes

An unsigned integer that specifies the attributes associated with the storage object. This value can be a combination of one or more of the following bitflags using a bitwise OR operation:

Value	Constant	Description
0	WEB_OBJECT_DEFAULT	Default object attributes. This value is used to indicate the object can be modified, or that the attributes for a previously existing object should not be changed.
1	WEB_OBJECT_NORMAL	A normal object that that can be read and modified by the application. This is the default attribute for new objects that are created by the application.
2	WEB_OBJECT_READONLY	A read-only object that can only be read by the application. Attempts to modify or replace the contents of the object will fail. Read-only objects

		can be deleted.
4	WEB_OBJECT_HIDDEN	A hidden object. Objects with this attribute are not returned when enumerated using the WebEnumObjects function. The object can only be accessed directly when specifying its label.

dwReserved

An unsigned integer value that is reserved for future use. This value should always be zero.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain additional information about the object. This parameter may be NULL if the information is not required.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebPutFileEx** function uploads the contents of a local file to the storage container specified by the *hStorage* handle. The caller is responsible for specifying the object's content type and the attributes for the object being created.

If a content type is provided, it must specify a valid MIME media type and subtype. For example, normal text files have a content type of **text/plain** while an XML-formatted text file would have a content type of **text/xml**. Files that contain unstructured binary data are typically identified as **application/octet-stream**.

If the label identifies an object that already exists in the container, and that object was created with the WEB_OBJECT_READONLY attribute, this function will fail. To replace a read-only object, the application must first move, rename or delete the existing object.

The **WebValidateLabel** function can be used to ensure a label is valid prior to calling this function. Refer to that function for more information about the difference between Windows file names and object labels

If you are uploading a large file and want your application to receive progress updates during the data transfer, use the **WebRegisterEvent** function and provide a pointer to a callback function that will receive event notifications.

The **WebUploadFile** and **WebUploadFileEx** functions provide similar functionality, but do not require a handle to an open storage container.

Example

```
WEB_STORAGE_OBJECT webObject;

// Upload a local file to the storage container, automatically
// determining the content type with normal read/write access
if (WebPutFileEx(hStorage, lpzLocalFile, lpzObjectLabel, NULL,
WEB_OBJECT_NORMAL, 0, &webObject))
{
    // The file was uploaded, display the object metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
}
```

```
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The file could not be uploaded, display the error
    TCHAR szError[128];

    WebGetErrorString(WebGetLastError(), szError, 128);
    _tprintf(_T("Unable to store \"%s\" (%s)\n"), lpszLocalFile, szError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebDownloadFile](#), [WebGetFile](#), [WebPutFile](#), [WebRegisterEvent](#), [WebUploadFile](#), [WebValidateLabel](#)

WebPutObject Function

```
BOOL WINAPI WebPutObject(  
    HSTORAGE hStorage,  
    LPCTSTR lpszObjectLabel,  
    LPCVOID lpvBuffer,  
    DWORD dwLength  
);
```

The **WebPutObject** function creates a new storage object or overwrites an existing object with the contents of the buffer provided.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A pointer to a null terminated string that specifies the label of the storage object that will be created or replaced. This parameter cannot be NULL or a zero-length string. The function will fail if the label contains any illegal characters.

lpvBuffer

A pointer to a buffer that contains the data to be stored. If this parameter is NULL, the *dwLength* parameter must have a value of zero and a storage object will be created which has no data associated with it.

dwLength

An unsigned integer value that specifies the number of bytes that will be copied from the *lpvBuffer* parameter and stored in the object. If the *lpvBuffer* parameter is NULL, this value must be zero or the function will fail.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebPutObject** function stores the contents of a buffer to the storage container specified by the *hStorage* handle. If an object with this name already exists, its data will be replaced with the contents of the file. The object will be created using the default attributes that permit read and write access, and it will automatically determine the content type based on the data being stored.

Although storage object labels are similar to Windows file names, they are case-sensitive. When requesting information about an object, your application must specify the label name exactly as it was created using this function. The object label cannot contain wildcard characters.

If the label identifies an object that already exists in the container, and that object was created with the WEB_OBJECT_READONLY attribute, this function will fail. To replace a read-only object, the application must first move, rename or delete the existing object.

If you want to specify the object content type, its attributes, or obtain additional information about the storage object that was created, use the **WebPutObjectEx** function.

If you want to upload the contents of a file, the **WebPutFile** function simplifies this process. The example code below demonstrates how the Windows API can be used to read from a local file and store the contents using **WebPutObject**.

If you are storing a large amount of data and want your application to receive progress updates during the data transfer, use the **WebRegisterEvent** function and provide a pointer to a callback function that will receive event notifications.

Example

```
HANDLE hFile = INVALID_HANDLE_VALUE;
LPBYTE lpContents = NULL;
DWORD dwLength = 0;
BOOL bFileRead = FALSE;

// Open a file on the local system and read the contents
// into a buffer that will be stored on the server
hFile = CreateFile(lpszLocalFile,
                  GENERIC_READ,
                  0,
                  NULL,
                  OPEN_EXISTING,
                  FILE_ATTRIBUTE_NORMAL,
                  NULL);

if (hFile == INVALID_HANDLE_VALUE)
{
    // Unable to open the file
    return;
}

// Get the size of the file and allocate a buffer large
// enough to store the contents of the file
dwLength = GetFileSize(hFile, NULL);
lpContents = (LPBYTE)LocalAlloc(LPTR, dwLength + 1);

if (lpContents == NULL)
{
    // Memory allocation failed
    return;
}

bFileRead = ReadFile(hFile, lpContents, dwLength, &dwLength, NULL);
CloseHandle(hFile);

if (!bFileRead)
{
    // Unable to read the contents of the file
    return;
}

// Store the contents of the buffer
if (WebPutObject(hStorage, lpszObjectLabel, lpContents, dwLength))
{
    _tprintf(_T("WebPutObject stored %lu bytes\n"), dwLength);
}
else
{
    // The object could not be created
    TCHAR szError[128];

    WebGetErrorString(WebGetLastError(), szError, 128);
    _tprintf(_T("Unable to create \"%s\" (%s)\n"), lpszObjectLabel, szError);
}
```

```
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebPutFile](#), [WebPutObjectEx](#), [WebRegisterEvent](#)

WebPutObjectEx Function

```
BOOL WINAPI WebPutObjectEx(  
    HSTORAGE hStorage,  
    LPCTSTR lpszObjectLabel,  
    LPCTSTR lpszContentType,  
    DWORD dwAttributes,  
    DWORD dwReserved,  
    LPCVOID lpvBuffer,  
    DWORD dwLength,  
    LPWEB_STORAGE_OBJECT lpObject  
);
```

The **WebPutObjectEx** function stores the contents of buffer to a container and returns information about the new object.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A pointer to a null terminated string that specifies the label of the storage object that will be created or replaced. This parameter cannot be NULL or a zero-length string. The function will fail if the label contains any illegal characters.

lpszContentType

A pointer to a null terminated string that identifies the contents of the buffer being stored. If this parameter is a NULL pointer, or specifies a zero-length string, the function will attempt to automatically determine the content type based on the object label and the contents of the buffer.

dwAttributes

An unsigned integer that specifies the attributes associated with the storage object. This value can be a combination of one or more of the following bitflags using a bitwise OR operation:

Value	Constant	Description
0	WEB_OBJECT_DEFAULT	Default object attributes. This value is used to indicate the object can be modified, or that the attributes for a previously existing object should not be changed.
1	WEB_OBJECT_NORMAL	A normal object that that can be read and modified by the application. This is the default attribute for new objects that are created by the application.
2	WEB_OBJECT_READONLY	A read-only object that can only be read by the application. Attempts to modify or replace the contents of the object will fail. Read-only objects can be deleted.
4	WEB_OBJECT_HIDDEN	A hidden object. Objects with this attribute are not returned when enumerated using the WebEnumObjects function. The object can only

	be accessed directly when specifying its label.
--	---

dwReserved

An unsigned integer value that is reserved for future use. This value should always be zero.

lpvBuffer

A pointer to a buffer that contains the data to be stored. If this parameter is NULL, the **dwLength** parameter must have a value of zero and a storage object will be created which has no data associated with it.

dwLength

An unsigned integer value that specifies the number of bytes that will be copied from the **lpvBuffer** parameter and stored in the object. If the **lpvBuffer** parameter is NULL, this value must be zero or the function will fail.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain additional information about the object that was created or replaced. This parameter may be NULL if the information is not required.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebPutObjectEx** function uploads the contents of a buffer to the storage container specified by the **hStorage** handle. The caller is responsible for specifying the object's content type and the attributes for the object being created.

Although storage object labels are similar to Windows file names, they are case-sensitive. When requesting information about an object, your application must specify the label name exactly as it was created using this function. The object label cannot contain wildcard characters.

If a content type is provided, it must specify a valid MIME media type and subtype. For example, typical text data has a content type of **text/plain** while XML-formatted text would have a content type of **text/xml**. Data which contains unstructured binary data is typically identified as **application/octet-stream**.

If the label identifies an object that already exists in the container, and that object was created with the **WEB_OBJECT_READONLY** attribute, this function will fail. To replace a read-only object, the application must explicitly move, rename or delete the existing object.

If you wish to store the contents of a null terminated string, you can use the **WebPutTextObject** function. If you want to upload the contents of a file, the **WebPutFileEx** function simplifies this process. The example code below demonstrates how the Windows API can be used to read from a local file and store the contents using **WebPutObjectEx**.

If you are storing a large amount of data and want your application to receive progress updates during the data transfer, use the **WebRegisterEvent** function and provide a pointer to a callback function that will receive event notifications.

Example

```
HANDLE hFile = INVALID_HANDLE_VALUE;  
LPBYTE lpContents = NULL;  
DWORD dwLength = 0;  
BOOL bFileRead = FALSE;
```

```

WEB_STORAGE_OBJECT webObject;

// Open a file on the local system and read the contents
// into a buffer that will be stored on the server
hFile = CreateFile(lpszLocalFile,
                  GENERIC_READ,
                  0,
                  NULL,
                  OPEN_EXISTING,
                  FILE_ATTRIBUTE_NORMAL,
                  NULL);

if (hFile == INVALID_HANDLE_VALUE)
{
    // Unable to open the file
    return;
}

// Get the size of the file and allocate a buffer large
// enough to store the contents of the file
dwLength = GetFileSize(hFile, NULL);
lpContents = (LPBYTE)LocalAlloc(LPTR, dwLength + 1);

if (lpContents == NULL)
{
    // Memory allocation failed
    return;
}

bFileRead = ReadFile(hFile, lpContents, dwLength, &dwLength, NULL);
CloseHandle(hFile);

if (!bFileRead)
{
    // Unable to read the contents of the file
    return;
}

// Store the contents of the buffer, identifying it as an unstructured
// stream of bytes, and the object is created with read/write access
if (WebPutObjectEx(hStorage,
                  lpszObjectLabel,
                  _T("application/octet-stream"),
                  WEB_OBJECT_NORMAL,
                  0,
                  lpContents,
                  dwLength,
                  &webObject))
{
    // The object was created or replaced, display the metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The object could not be created

```

```
TCHAR szError[128];

WebGetErrorString(WebGetLastError(), szError, 128);
_tprintf(_T("Unable to create \"%s\" (%s)\n"), lpszObjectLabel, szError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebPutFileEx](#), [WebPutObject](#), [WebPutTextObject](#), [WebRegisterEvent](#)

WebPutTextObject Function

```
BOOL WINAPI WebPutTextObject(  
    HSTORAGE hStorage,  
    LPCTSTR lpszObjectLabel,  
    LPCTSTR lpszObjectText,  
    INT cchObjectText,  
    LPWEB_STORAGE_OBJECT lpObject,  
);
```

The **WebPutTextObject** function creates or replaces a text object with the contents of a string buffer. Additional information about the object is returned to the caller.

Parameters

hStorage

A handle to the storage container.

lpszObjectLabel

A pointer to a null terminated string that specifies the label of the text object that should be created or updated.

lpszObjectText

A pointer to a string which contains the text that should be uploaded to the storage server. This parameter may be NULL, in which case the object is created but no data is stored.

cchObjectText

The number of characters that are stored in the *lpszObjectText* string. If this value is -1, the size of the string will be determined automatically by counting the number of characters up to the terminating null character. If the *lpszObjectText* parameter is NULL, this value must be zero.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain additional information about the object. This parameter may be NULL if the information is not required.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebPutTextObject** function is used to store the contents of a string as a text object. The string should be terminated with a null character and cannot contain embedded nulls. If the Unicode version of this function is called, the contents of the string will be normalized prior to being converted to UTF-8 using canonical composition, where decomposed characters are combined to create their canonical precomposed equivalent.

The size of the text object that is created can be different from the length of the string buffer passed to this function, depending on how the text has been encoded and stored. This will almost always be the case when the Unicode version of this function is called because the UTF-16 string will be converted and stored as UTF-8 encoded text. This encoding will typically increase the size of the stored object unless the string only contains ASCII characters.

This function should only be used to store textual data in a null terminated string, and should never be used to store binary data. If you wish to create or update an object which contains binary data, use the **WebPutObjectEx** function instead. If you want to upload the contents of a file, the

WebPutFileEx function simplifies this process.

Additional metadata about the object will be returned in the [WEB_STORAGE_OBJECT](#) structure provided by the caller, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents.

Example

```
LPCTSTR lpszObjectLabel = _T("LoremIpsum");
LPCTSTR lpszObjectText = _T("Lorem ipsum dolor sit amet, consectetur adipiscing elit.");
WEB_STORAGE_OBJECT webObject;

if (WebPutTextObject(hStorage, lpszObjectLabel, lpszObjectText, -1, &webObject))
{
    // The object was created or replaced, display the metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    DWORD dwError = WebGetLastError();
    TCHAR szError[128];

    WebGetErrorString(dwError, szError, 128);
    _ftprintf(stderr, _T("Unable to store \"%s\": %s\n"), lpszObjectLabel, szError);
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebCompareText](#), [WebGetObjectEx](#), [WebGetTextObject](#), [WebPutFileEx](#), [WebPutObjectEx](#)

WebRegisterAppId Function

```
BOOL WINAPI WebRegisterAppId(  
    LPCTSTR LpszAppId  
);
```

The **WebRegisterAppId** function registers a unique application identifier with the server.

Parameters

lpszAppId

A pointer to a null terminated string which identifies the application requesting access. This parameter cannot be NULL or point to a zero-length string. If the application ID contains illegal characters, the function will fail. See the remarks below on the recommended method for identifying your application.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebRegisterAppId** function registers an application ID with the server which uniquely identifies the application that is requesting access to the storage container. The ID must only consist of ASCII letters, numbers, the period and underscore character. Whitespace characters and non-ASCII Unicode characters are not permitted. The maximum length of an application ID string is 64 characters, including the terminating null character.

It is recommended that you use a standard format for the application ID that consists of your company name, application name and optionally a version number. For example:

- MyCompany.MyApplication
- MyCompany.MyApplication.1

It is important to note that with these two example IDs, although they are similar, they reference two different applications. Objects stored using the first ID will not be accessible using the second ID. If you want to store objects that should be shared between all versions of the application, it is recommended that you use the first form, without the version number. If you want to store objects that should only be accessible to a specific version of your application, then it is recommended that you use the second form that includes the version number.

It is safe to call this function with an application ID that was previously registered. If the provided application ID has already been registered, this function will succeed. You can choose to call this function every time your application starts to ensure the application has been registered correctly.

If you no longer wish to use an application ID you have previously registered, you can call the **WebUnregisterAppId** function. Exercise caution when unregistering an application. This will cause all objects stored using that ID to be deleted by the storage server. Once an application ID has been unregistered, the operation is permanent. Calling **WebUnregisterAppId** and then **WebRegisterAppId** again using the same ID will force the system to create new access tokens for your application. You will not be able to regain access to the objects that were previously stored using that ID.

The application ID is intended to be an application defined human-readable string that uniquely identifies your application. If you want to obtain the internal storage ID associated with your

application, use the **WebGetStorageId** function. The storage ID is a fixed-length string of letters and numbers guaranteed to be unique across all applications that you register.

It is not required for your application to create a unique application ID. Each storage account has a default internal application ID named **SocketTools.Storage.Default**. This default ID is used if a NULL pointer or an empty string is specified to functions like **WebOpenStorage**. It is intended to identify storage available to all applications that you create.

To enumerate the application IDs that you have created, use the **WebGetFirstApplication** and **WebGetNextApplication** functions.

Example

```
HSTORAGE hStorage = INVALID_HANDLE;
LPCTSTR lpszAppId = _T("MyCompany.WebTest.1");
LPCTSTR lpszLocalFile = _T("TestDocument.pdf");
LPCTSTR lpszObjectLabel = _T("TestDocument.pdf");
WEB_STORAGE_OBJECT webObject = { 0, };
BOOL bSuccess = FALSE;

// Register the application ID which identifies the application
if (!WebRegisterAppId(lpszAppId))
{
    tprintf(_T("Unable to register the application ID"));
    _exit(0);
}

// Open the application storage container
hStorage = WebOpenStorage(lpszAppId, WEB_STORAGE_GLOBAL);

if (hStorage == INVALID_HANDLE)
{
    tprintf(_T("Unable to open global storage for this application"));
    _exit(0);
}

// Upload a local file to the storage server and return information
// about the stored object when it completes
bSuccess = WebPutFileEx(hStorage,
                        lpszLocalFile,
                        lpszObjectLabel,
                        NULL,
                        WEB_OBJECT_DEFAULT,
                        0,
                        &webObject);

if (bSuccess)
{
    // Print information about the object that was created
    _tprintf(_T("Object: %s\n"), webObject.szObjectId);
    _tprintf(_T("Label: %s\n"), webObject.szLabel);
    _tprintf(_T("Size: %I64u\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest: %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The upload failed, display error information
    DWORD dwError = WebGetLastError();
}
```

```
TCHAR szError[128];

WebGetErrorString(dwError, szError, 128);
_tprintf(_T("WebPutFileEx failed, error 0x%08lX (%s)\n"), dwError, szError);
}

// Release the handle allocated for this storage session
WebCloseStorage(hStorage);
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebCloseStorage](#), [WebOpenStorage](#), [WebGetFirstApplication](#), [WebGetNextApplication](#),
[WebGetStorageId](#), [WebUnregisterAppId](#), [WebValidateAppId](#)

WebRegisterEvent Function

```
INT WINAPI WebRegisterEvent(  
    HSTORAGE hStorage,  
    UINT nEventId,  
    WEBEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

The **WebRegisterEvent** function registers an event handler for the specified event.

Parameters

hStorage

A handle to the storage container.

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. This parameter cannot be zero. The following values may be used:

Constant	Description
WEB_EVENT_CONNECT (1)	The connection to the storage server has been established and the session has been authenticated. This is the first event that occurs when initiating an operation to create or retrieve a storage object.
WEB_EVENT_DISCONNECT (2)	The connection to the storage server has been closed and the session is terminating. This is the last event that occurs after completing an operation to create or retrieve a storage object.
WEB_EVENT_READ (4)	The contents of an object is being read from the storage container. This event occurs only once after the operation has been initiated by the application. This event can also be generated if there is an error during the process of retrieving the object contents from the container.
WEB_EVENT_WRITE (8)	The contents of an object is being written to the storage container. This event occurs only once after the operation has been initiated by the application. This event can also be generated if there is an error during the process of submitting the object contents to the container.
WEB_EVENT_TIMEOUT (16)	The operation has exceeded the specified timeout period. The application may attempt to retry the operation or report an error to the user. This event typically indicates a connectivity problem with the storage server.
WEB_EVENT_CANCEL (32)	The operation has been canceled. This event occurs after the application calls WebCancelTransfer while an object is being stored or retrieved.
WEB_EVENT_PROGRESS (64)	A storage operation is in progress. This event periodically occurs as the contents of a storage object is being read or written from the container. To retrieve information about the status of the operation, the application should register a

handler for this event and call the WebGetTransferStatus function from within that handler.
--

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **WebEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function specified by *lpEventProc*. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebRegisterEvent** function associates a callback function with a specific event. The event handler is a **WebEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the storage handle, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

Events are only generated as the result of a call to the **WebGetFile**, **WebGetFileEx**, **WebPutFile** and **WebPutFileEx** functions. Other storage functions will not generate event notifications.

This function is typically used to register an event handler that is invoked while the contents of a storage object is being transferred. The **WEB_EVENT_PROGRESS** event will only be generated periodically during the transfer to ensure the application is not flooded with event notifications. It is guaranteed that at least one **WEB_EVENT_PROGRESS** notification will occur at the beginning of the transfer, and one at the end of the transfer when it has completed.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cswebv10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebCancelTransfer](#), [WebEventProc](#), [WebGetTransferStatus](#)

WebRenameObject Function

```
BOOL WINAPI WebRenameObject(  
    HSTORAGE hStorage,  
    LPCTSTR lpszOldLabel,  
    LPCTSTR lpszNewLabel  
);
```

The **WebRenameObject** function changes the label associated with the storage object.

Parameters

hStorage

A handle to the storage container.

lpszOldLabel

A pointer to a null terminated string which specifies the name of the existing storage object to be renamed. This parameter must specify a valid object label and cannot be a NULL pointer or an empty string.

lpszNewLabel

A pointer to a null terminated string which specifies a new label for the storage object being moved. This parameter may not be NULL or point to an empty string.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebRenameObject** function is used to change the label for an existing storage object. Although storage object labels are similar to Windows file names, they are case-sensitive. When renaming an object, your application must specify the original label name exactly as it was created. The object label cannot contain wildcard characters.

To duplicate an existing storage object, use the **WebCopyObject** function. If you need to move the object to a different storage container, use the **WebMoveObject** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebCopyObject](#), [WebDeleteObject](#), [WebMoveObject](#)

WebResetStorage Function

```
BOOL WINAPI WebResetStorage(  
    HSTORAGE hStorage  
);
```

The **WebResetStorage** function resets the application storage container and deletes all stored objects.

Parameters

hStorage

A handle to the storage container.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The storage container contains information for each of the objects that have been stored using the handle returned by the **WebOpenStorage** function. Each of these objects are associated with both the application ID and the storage type that was specified by the caller. This function instructs the server to reset the container back to its initial state, deleting all of the objects that were stored in it.



Exercise caution when using this function. The reset operation is immediate and the objects that are stored in the container are permanently deleted. They cannot be recovered by your application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebDeleteObject](#), [WebOpenStorage](#), [WebRegisterAppId](#), [WebUnregisterAppId](#)

WebResetStorageEx Function

```
BOOL WINAPI WebResetStorageEx(  
    HSTORAGE hStorage,  
    DWORD dwStorageType,  
    LPVOID LpReserved,  
    LPDWORD LpdwObjects  
);
```

The **WebResetStorageEx** function resets the specified storage container and deletes all stored objects.

Parameters

hStorage

A handle to the storage container.

dwStorageType

An integer value that identifies the storage container type. One of the following values should be specified:

Constant	Description
WEB_STORAGE_GLOBAL (1)	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN (2)	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
WEB_STORAGE_MACHINE (3)	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER (4)	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

LpReserved

A reserved parameter that should always be NULL. If your project targets the x64 platform, this must be a 64-bit pointer with a value of zero. If this parameter is not a NULL pointer, the function will fail.

lpdwObjects

A pointer to an unsigned integer value which will contain the number of objects that were deleted when the contents of the container were expunged.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The storage container contains information for each of the objects that have been stored using the handle returned by the **WebOpenStorage** function. Each of these objects are associated with both the application ID and the storage type that was specified by the caller. This function instructs the server to reset the container back to its initial state, deleting all of the objects that were stored in it.



Exercise caution when using this function. The reset operation is immediate and the objects that are stored in the container are permanently deleted. They cannot be recovered by your application.

Unlike the **WebResetStorage** function which deletes all of the objects in the container opened using **WebOpenStorage**, this function enables you reset the contents of a different container without explicitly opening it. For example, if you used **WebOpenStorage** to open the WEB_STORAGE_GLOBAL container, you could call this version of the function using that handle, but specifying the WEB_STORAGE_USER container to remove all of the objects created by the current user without affecting the stored objects in the global container.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebDeleteObject](#), [WebOpenStorage](#), [WebRegisterAppId](#), [WebResetStorage](#), [WebUnregisterAppId](#)

WebSetLastError Function

```
VOID WINAPI WebSetLastError(  
    DWORD dwErrorCode  
);
```

The **WebSetLastError** function sets the last error code for the current thread. This function is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or WEB_ERROR. Those functions which call **WebSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **WebGetLastError** function. The use of **WebGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswebv10.lib

See Also

[WebGetErrorString](#), [WebGetLastError](#)

WebSetStorageTimeout Function

```
INT WINAPI WebSetStorageTimeout(  
    HSTORAGE hStorage,  
    INT nSeconds  
);
```

The **WebSetStorageTimeout** function sets the number of seconds the client will wait for a response from the storage server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

nSeconds

The number of seconds to wait for a storage operation to complete.

Return Value

If the function succeeds, the return value is the previous timeout period set for this storage handle. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The default timeout period is 10 seconds. The minimum timeout period is 5 seconds, and the maximum timeout period is 60 seconds. Values outside of this range will be normalized internally and will not cause the function to fail.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

See Also

[WebGetStorageTimeout](#), [WebOpenStorage](#)

WebUninitialize Function

```
VOID WINAPI WebUninitialize();
```

The **WebUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **WebInitialize** call before it can call any of the other library functions. When it has completed the use of library, the application must call **WebUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There must be a call to **WebUninitialize** for every successful call to **WebInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

See Also

[WebCloseStorage](#), [WebInitialize](#), [WebOpenStorage](#), [WebRegisterAppId](#)

WebUnregisterAppId Function

```
BOOL WINAPI WebUnregisterAppId(  
    LPCTSTR lpszAppId  
);
```

The **WebUnregisterAppId** function unregisters the application identifier and deletes all associated storage objects.

Parameters

lpszAppId

A pointer to a null terminated string which specifies the application ID to be deleted. This parameter cannot be NULL or point to a zero-length string. If the application ID contains illegal characters, the function will fail.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebUnregisterAppId** function deletes the internal storage identifier associated with the application ID and revokes all access tokens that were granted for the application. This operation is immediate and permanent.



Exercise caution when using this function. This will permanently delete all objects that were stored for the specified application. Calling **WebUnregisterAppId** and then **WebRegisterAppId** again using the same ID will force the system to create new access tokens for your application. You will not be able to regain access to the objects that were previously stored using that ID.

This function cannot be used to unregister the default storage application identifier **SocketTools.Storage.Default**. If this ID is specified, the function will fail with an error indicating that the ID is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebOpenStorage](#), [WebRegisterAppId](#), [WebResetStorage](#), [WebValidateAppId](#)

WebUploadFile Function

```
BOOL WINAPI WebUploadFile(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszObjectLabel,  
    LPWEB_STORAGE_OBJECT lpObject  
);
```

The **WebUploadFile** function uploads the contents of a local file and creates or overwrites a storage object.

Parameters

lpszLocalFile

A pointer to a null terminated string that specifies the name of the local file that will be created or overwritten with the contents of the storage object. If a path is not specified, the file will be created in the current working directory.

lpszObjectLabel

A pointer to a null terminated string that specifies the label of the object that should be retrieved from the server. This parameter cannot be NULL or an empty string. Refer to the **WebValidateLabel** function for more information about object labels.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain additional information about the object that has been downloaded. This parameter may be NULL if the information is not required.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebUploadFile** function uploads the contents of a local file and either creates a new storage object, or replaces an object if one already exists with the same label. The object will be created in the storage container `WEB_STORAGE_GLOBAL` using the application ID

SocketTools.Storage.Default. This is the same default container used by the **WebDownloadFile** function.

If the label identifies an object that already exists in the container, and that object was created with the `WEB_OBJECT_READONLY` attribute, this function will fail. To replace a read-only object, the application must explicitly move, rename or delete the existing object.

Unlike the **WebPutFile** and **WebPutFileEx** functions, it is not required that you open a storage container using the **WebOpenStorage** function prior to calling **WebUploadFile**.

Additional metadata about the object will be returned in the `WEB_STORAGE_OBJECT` structure provided by the caller, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents.

The **WebUploadFileEx** function provides similar functionality with a more complex interface that supports custom application IDs, additional options and the ability to provide a callback function that is invoked during the upload process.

Example

```

WEB_STORAGE_OBJECT webObject;

// Upload a local file to the global storage container
if (WebUploadFile(lpszLocalFile, lpszObjectLabel, &webObject))
{
    // The object was uploaded, display the metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The object could not be uploaded, display the error
    TCHAR szError[128];

    WebGetErrorString(WebGetLastError(), szError, 128);
    _tprintf(_T("Unable to upload \"%s\" (%s)\n"), lpszLocalFile, szError);
}

```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebDownloadFile](#), [WebGetFile](#), [WebPutFile](#), [WebUploadFileEx](#), [WebValidateLabel](#)

WebUploadFileEx Function

```
BOOL WINAPI WebUploadFileEx(  
    LPCTSTR lpszLocalFile,  
    LPCTSTR lpszObjectLabel,  
    LPCTSTR lpszAppId,  
    DWORD dwStorageType,  
    DWORD dwAttributes,  
    DWORD dwReserved,  
    DWORD dwTimeout,  
    LPWEB_STORAGE_OBJECT lpObject,  
    WEBEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

The **WebUploadFileEx** function uploads the contents of a local file and creates or overwrites a storage object.

Parameters

lpszLocalFile

A pointer to a null terminated string that specifies the name of the local file that will be created or overwritten with the contents of the storage object. If a path is not specified, the file will be created in the current working directory.

lpszObjectLabel

A pointer to a null terminated string that specifies the label of the object that should be retrieved from the server.

lpszAppId

A pointer to null terminated string which specifies the application ID for the storage container. The application ID is a string that uniquely identifies the application and can only contain letters, numbers, the period and the underscore character. If this parameter is NULL or an empty string, the default identifier **SocketTools.Storage.Default** will be used.

dwStorageType

An integer value that identifies the storage container type. One of the following values should be specified:

Constant	Description
WEB_STORAGE_GLOBAL (1)	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN (2)	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.

WEB_STORAGE_MACHINE (3)	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER (4)	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

dwAttributes

An unsigned integer that specifies the attributes associated with the storage object. This value can be a combination of one or more of the following bitflags using a bitwise OR operation:

Value	Constant	Description
0	WEB_OBJECT_DEFAULT	Default object attributes. This value is used to indicate the object can be modified, or that the attributes for a previously existing object should not be changed.
1	WEB_OBJECT_NORMAL	A normal object that that can be read and modified by the application. This is the default attribute for new objects that are created by the application.
2	WEB_OBJECT_READONLY	A read-only object that can only be read by the application. Attempts to modify or replace the contents of the object will fail. Read-only objects can be deleted.
4	WEB_OBJECT_HIDDEN	A hidden object. Objects with this attribute are not returned when enumerated using the WebEnumObjects function. The object can only be accessed directly when specifying its label.

dwReserved

An unsigned integer value that is reserved for future use. This value should always be zero.

dwTimeout

An unsigned integer value that specifies a timeout period in seconds. If this value is zero, a default timeout period will be used.

lpObject

A pointer to a [WEB_STORAGE_OBJECT](#) structure that will contain additional information about the object that has been downloaded. This parameter may be NULL if the information is not required.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **WebEventProc** callback function. If this parameter is NULL, no callback function will be invoked during the data transfer.

dwParam

A user-defined integer value that is passed to the callback function specified by *lpEventProc*. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebUploadFileEx** function uploads the contents of a local file and either creates a new storage object, or replaces an object if one already exists with the same label. The **WebUploadFile** function provides a simpler interface that defaults to uploading an object to the global storage container.

If the label identifies an object that already exists in the container, and that object was created with the `WEB_OBJECT_READONLY` attribute, this function will fail. To replace a read-only object, the application must explicitly move, rename or delete the existing object.

Unlike the **WebPutFile** and **WebPutFileEx** functions, it is not required that you open a storage container using the **WebOpenStorage** function prior to calling **WebUploadFileEx**.

Additional metadata about the object will be returned in the `WEB_STORAGE_OBJECT` structure provided by the caller, such as the date and time the object was created, the content type and the SHA-256 hash of the object contents.

If you are uploading a large file and want your application to receive progress updates during the data transfer, provide a pointer to a callback function as the *lpEventProc* parameter. That function will receive event notifications as the data is being uploaded.

Example

```
WEB_STORAGE_OBJECT webObject;

// Upload a local file to the global storage container
if (WebUploadFileEx(lpszLocalFile,
                   lpszObjectLabel,
                   lpszAppId,
                   WEB_STORAGE_GLOBAL,
                   WEB_OBJECT_DEFAULT,
                   0,
                   &webObject,
                   NULL, 0))
{
    // The object was uploaded, display the metadata
    _tprintf(_T("Object:  %s\n"), webObject.szObjectId);
    _tprintf(_T("Label:   %s\n"), webObject.szLabel);
    _tprintf(_T("Size:    %lu\n"), webObject.dwObjectSize);
    _tprintf(_T("Digest:  %s\n"), webObject.szDigest);
    _tprintf(_T("Content: %s\n"), webObject.szContent);
}
else
{
    // The object could not be uploaded, display the error
    TCHAR szError[128];
```

```
WebGetErrorString(WebGetLastError(), szError, 128);  
_tprintf(_T("Unable to upload \"%s\" (%s)\n"), lpszLocalFile, szError);  
}
```

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebDownloadFile](#), [WebGetFileEx](#), [WebPutFileEx](#), [WebUploadFile](#)

WebValidateAppId Function

```
BOOL WINAPI WebValidateAppId(  
    LPCTSTR LpszAppId  
);
```

The **WebValidateAppId** function validates the specified application identifier.

Parameters

lpszAppId

A pointer to a null terminated string which specifies the application ID to be validated. This parameter cannot be NULL or point to a zero-length string. If the application ID contains illegal characters, the function will fail.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

The **WebValidateAppId** function is used to determine if the specified application identifier is valid and has been previously registered using the **WebRegisterAppId** function. The ID must only consist of ASCII letters, numbers, the period and underscore character. Whitespace characters and non-ASCII Unicode characters are not permitted. The maximum length of an application ID string is 64 characters, including the terminating null character.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstdlib10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebRegisterAppId](#), [WebUnregisterAppId](#)

WebValidateLabel Function

```
BOOL WINAPI WebValidateLabel(  
    LPCTSTR lpszObjectLabel  
);
```

The **WebValidateLabel** function validates the specified object label.

Parameters

lpszObjectLabel

A pointer to a null terminated string which specifies the object label to be validated. This parameter cannot be NULL or point to a zero-length string.

Return Value

If the function succeeds, the return value is a non-zero. If the function fails, the return value is zero. To get extended error information, call **WebGetLastError**.

Remarks

Object labels are similar to Windows file names, except they are case-sensitive. The maximum length of a label string is 512 characters, including the terminating null character. Leading and trailing whitespace (spaces, tabs, linebreaks, etc.) are ignored in label names.

Illegal characters include ASCII and Unicode control characters 1 through 31, single quotes (39), double quotes (34), less than symbol (60), greater than symbol (62), pipe (124), asterisk (42) and question mark (63). A null character (0) specifies the end of the label and any subsequent characters are ignored. It is not possible to embed null characters in the label name.

Label names may contain forward slash (47) characters and backslash (92) characters, however it is important to note that objects are not stored in a hierarchical structure. An application can create its own folder-like structure to the labels it creates, but this structure is not imposed or enforced by the library.

If the application is built to use Unicode, labels can contain Unicode characters which are internally encoded as UTF-8. This is important to consider if you have an project built using a multi-byte (ANSI) character set and it needs to access an object that was created using Unicode characters. In that case, the ANSI application must be prepared to handle UTF-8 encoded names and display them appropriately.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstoools10.h

Import Library: cswebv10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WebRegisterAppId](#), [WebUnregisterAppId](#)

Web Services Data Structures

- INITDATA
- SYSTEMTIME
- WEB_LOCATION
- WEB_STORAGE_APPLICATION
- WEB_STORAGE_QUOTA
- WEB_STORAGE_OBJECT
- WEB_STORAGE_TRANSFER

INITDATA Structure

This structure contains information about the SocketTools library when the initialization function returns.

```
typedef struct _INITDATA
{
    DWORD        dwSize;
    DWORD        dwVersionMajor;
    DWORD        dwVersionMinor;
    DWORD        dwVersionBuild;
    DWORD        dwOptions;
    DWORD_PTR    dwReserved1;
    DWORD_PTR    dwReserved2;
    TCHAR        szDescription[128];
} INITDATA, *LPINITDATA;
```

Members

dwSize

Size of this structure. This structure member must be set to the size of the structure prior to calling the initialization function. Failure to do so will cause the function to fail.

dwVersionMajor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the major version number for the library.

dwVersionMinor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the minor version number for the library.

dwVersionBuild

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the build number for the library.

dwOptions

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved1

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved2

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

szDescription

A null terminated string which describes the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

SYSTEMTIME Structure

The **SYSTEMTIME** structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME
{
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *LPSYSTEMTIME;
```

Members

wYear

Specifies the current year. The year value must be greater than 1601.

wMonth

Specifies the current month. January is 1, February is 2 and so on.

wDayOfWeek

Specifies the current day of the week. Sunday is 0, Monday is 1 and so on.

wDay

Specifies the current day of the month

wHour

Specifies the current hour.

wMinute

Specifies the current minute

wSecond

Specifies the current second.

wMilliseconds

Specifies the current millisecond.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include windows.h.

WEB_LOCATION Structure

The **WEB_LOCATION** structure contains information about the current location of the local computer system.

```
typedef struct _WEB_LOCATION
{
    TCHAR    szLocationId[40];
    TCHAR    szIPAddress[46];
    UINT     nAutonomousSystemNumber;
    TCHAR    szOrganization[64];
    TCHAR    szRegionName[64];
    UINT     nRegionCode;
    TCHAR    szCountryName[64];
    TCHAR    szCountryAlpha2[4];
    TCHAR    szCountryAlpha3[4];
    UINT     nCountryCode;
    TCHAR    szSubdivision[64];
    TCHAR    szSubdivisionCode[4];
    TCHAR    szCityName[64];
    TCHAR    szPostalCode[32];
    TCHAR    szCoordinates[32];
    TCHAR    szTimezone[64];
    TCHAR    szTzShortName[16];
    LONG     nTimezoneOffset;
    DOUBLE   dLatitude;
    DOUBLE   dLongitude;
    SYSTEMTIME stLocalTime;
} WEB_LOCATION, *LPWEB_LOCATION;
```

Members

szLocationId

A null-terminated string which contains a string of hexadecimal characters which uniquely identifies the location for this computer system. This value is used internally by the location service, and may also be used by the application for its own purposes. If this value changes in subsequent queries, it indicates the external IP address for the local system has changed.

lpszIPAddress

A null-terminated string which contains the external IP address for the local system. If the system has been assigned multiple IP addresses, it reflects the address of the interface that was used to establish a connection with the SocketTools server. If the connection is made through a Virtual Private Network (VPN) it will use that assigned IP address. If a connection is made through a proxy server, the IP address may be the address of the proxy rather than the local host, depending on how the connection is made.

nAutonomousSystemNumber

An unsigned integer value which is used to uniquely identify a global network (autonomous system) that is connected to the Internet. This number is assigned by regional registries and used by large networks, such as Internet Service Providers, for exchanging routing information with one another. This value can be used to determine the ownership of a particular network.

szOrganization

A null-terminated string which identifies the organization associated with the local system's external IP address. For residential end-users this is typically the name of their Internet Service

provider, however it may also identify a private company such as Microsoft, Google or Amazon. Because of the nature of how this information is updated, the organization names can change over time due to acquisitions or changes of ownership. If the owner of the network cannot be determined, this member may contain an empty string.

szRegionName

A null-terminated string which identifies the region in which the external IP address is located. This refers to a broad geographical area, such as "North America" or "Southeast Asia" and uses the conventions for supranational regions as defined by UN M49 codes. These names will always be in English, regardless of the current system locale.

nRegionCode

An unsigned integer value which identifies the geographical region in which the external IP address is located. This value corresponds to the name returned in the ***szRegionName*** structure member. The numeric values use the UN M49 standard established by the United Nations Statistics Division.

szCountryName

A null-terminated string which contains the full name of the country in which the external IP address is located. These names will always be in English, regardless of the current system locale.

szCountryAlpha2

A null-terminated string which contains the ISO 3166-1 alpha-2 code for the country the external IP address is located in. This is a two-letter country code established by the International Organization for Standardization (ISO). For example, the code for the United States is "US".

szCountryAlpha3

A null-terminated string which contains the ISO 3166-1 alpha-3 code for the country the external IP address is located in. This is a three-letter country code established by the International Organization for Standardization (ISO). For example, the code for the United States is "USA".

nCountryCode

An unsigned integer value which identifies the country where the external IP address is located. These codes can be up to three digits (usually displayed with leading zeros as necessary) and correspond to the country codes assigned by the United Nations. For example, the code for the United States is 840. It is important to note that these are not international dialing codes and should not be used with telephony applications.

szSubdivision

A null-terminated string which identifies the geopolitical subdivision within a country where the external IP address is located. In the United States, this will contain the full name of the state or commonwealth. In Canada, this will contain the name of the province or territory. These names will always be in English, regardless of the current system locale.

szSubdivisionCode

A null-terminated string which is either a two- or three-letter code which identifies a geopolitical subdivision within the country where the external IP address is located. These codes are defined by the ISO 3166-2 standard. For example, the code for the state of California in the United States is "CA". For specificity, these subdivision codes are often combined with the ISO 3166-1 alpha-2 or alpha-3 code for the country the subdivision is located within. Using the previous example of California, ISO alpha-2 code and subdivision code would be combined as "US-CA".

szCityName

A null-terminated string which identifies the city in which the external IP address is located. These names will always be in English, regardless of the current system locale. If the city name cannot be determined, this member may contain an empty string.

szPostalCode

A null-terminated string which contains the postal code associated with the area where the IP address is located. In the United States, this is a 5-digit numeric code. In Canada, this will contain the forward sortation area (the first three characters of the six character postal code). Local delivery portions of a postal code (such as the ZIP+4 code in the United States, or the local delivery unit in Canada) are not included. This information will be most accurate within the geographic region of North America. Postal codes are locale specific, and this structure member may contain an empty string if the postal code for the location cannot be determined.

szCoordinates

A null-terminated string which specifies the location expressed using the Universal Transverse Mercator (UTM) coordinate system with the WGS-84 ellipsoid. UTM coordinates are commonly used with the Global Positioning System (GPS) and are comprised of three parts: the zone, the easting (the eastward-measured distance or *x*-coordinate) and the northing (the northward-measured distance or *y*-coordinate). An example of a string value returned in this structure member would be "14S 702089E 3646476N".

szTimezone

A null-terminated string which specifies the full time zone name in which the external IP address is located. These names are defined by the Internet Assigned Numbers Authority (IANA) and have values like "America/Los_Angeles" and "Europe/London". These time zones may also be defined as "Etc/GMT+10" if there is not a regional name associated with the time zone.

szTzShortName

A null-terminated string which specifies the abbreviated time zone code in which the external IP address is located. If daylight savings time is used within the time zone, then this value can change based on whether or not daylight savings is in effect. For example, if the IP address is located within the Pacific time zone in the United States, this will return "PDT" when daylight savings is in effect and "PST" when it is not. If a short time zone code cannot be determined, a value such as "UTC+9" may be returned, indicating the number of hours ahead or behind UTC.

nTimezoneOffset

A signed integer which specifies the number of seconds east or west of the prime meridian (UTC). A positive value indicates a time zone that is east of the prime meridian and a negative value indicates a time zone that is west of the prime meridian. For example, the Pacific time zone in the western United States during daylight savings time would have a value of -25200 (7 hours).

dLatitude

A floating point value which specifies the latitude of the location in decimal format. A positive value indicates a location that is north of the equator, while a negative value is a location that is south of the equator.

dLongitude

A floating point value which specifies the longitude of the location in decimal format. A positive value indicates a location that is east of the prime meridian, while a negative value is a location that is west of the prime meridian.

stLocalTime

A [SYSTEMTIME](#) structure that contains information about the current date and time at the location, adjusted for its time zone and whether or not it's in daylight savings time.

Remarks

Use the **WebGetLocation** function to populate this structure with information about the physical location of the local computer system.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cstools10.h`

See Also

[WebGetLocation](#)

WEB_STORAGE_APPLICATION Structure

The **WEB_STORAGE_APPLICATION** structure contains information about a registered application.

```
typedef struct _WEB_STORAGE_APPLICATION
{
    TCHAR        szAppId[64];
    TCHAR        szApiKey[64];
    TCHAR        szLuid[64];
    DWORD        dwTokens;
    DWORD        dwReserved;
    SYSTEMTIME   stCreated;
    SYSTEMTIME   stUpdated;
} WEB_STORAGE_APPLICATION, *LPWEB_STORAGE_APPLICATION;
```

Members

szAppId

A null-terminated string which contains the application ID that was registered using the **WebRegisterAppId** function.

szApiKey

A null-terminated string which contains the API key value that is used internally by the storage services. The AppID is effectively the human readable alias for this key value used to identify the stored objects for the application. This value is guaranteed to be unique across all applications registered with the storage service.

szLuid

A null-terminated string which specifies a locally unique value associated with the registered application. This value is used internally by the storage service and guaranteed to be unique to the storage account that has registered the application. Note that there are no functions that currently accept the application LUID as parameter, but you may choose to use this value in your own application for other purposes.

dwTokens

An unsigned integer value which specifies the number of access tokens associated with the registered application. Typically there is only one access token associated with a given AppID at any one time, although it is possible that the API may allocate additional access tokens under some circumstances.

dwReserved

An unsigned integer value that is reserved for future use. This value will always be zero.

stCreated

A SYSTEMTIME structure that specifies the date and time the AppID was registered. This value is represented using Coordinated Universal Time (UTC) and is not adjusted for the local time zone.

stUpdated

A SYSTEMTIME structure that specifies the date and time the AppID object was last updated. This value is represented using Coordinated Universal Time (UTC) and is not adjusted for the local time zone. When a storage object is first created, this value will be the same as the object creation time.

Remarks

This structure is used in conjunction with the **WebEnumApplications** function which is used to

enumerate all of the registered applications for the storage account. It is also used with the **WebGetFirstApplication** and **WebGetNextApplication** functions which provide an alternative method for enumerating registered applications.

To adjust the creation and update times to account for the local time zone, use the **SystemTimeToTzSpecificLocalTime** function. If you prefer to use FILETIME values, use the **SystemTimeToFileTime** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include ctools10.h

Unicode: Implemented as Unicode and ANSI versions.

WEB_STORAGE_OBJECT Structure

The **WEB_STORAGE_OBJECT** structure contains information about an individual storage object.

```
typedef struct _WEB_STORAGE_OBJECT
{
    TCHAR        szObjectId[64];
    TCHAR        szLabel[512];
    TCHAR        szDigest[128];
    TCHAR        szContent[128];
    DWORD        dwAttributes;
    DWORD        dwObjectSize;
    SYSTEMTIME   stCreated;
    SYSTEMTIME   stModified;
} WEB_STORAGE_OBJECT, *LPWEB_STORAGE_OBJECT;
```

Members

szObjectId

A null-terminated string which contains the unique identifier associated with this object. Object IDs are guaranteed to be unique for each storage object that is created by the application. The maximum length of an object ID is 64 characters, including the terminating null character.

szLabel

A null-terminated string which contains the label assigned to the object by the application. Object labels are case-sensitive and must be unique for each object. An application uses labels to reference an object with a human-recognizable name, rather than referencing them by their object ID. The maximum length of an object label is 512 characters, including the terminating null character.

szDigest

A null-terminated string which specifies the digest of the object contents, computed using an SHA-256 hash. The maximum length of the *szDigest* string is 128 characters, including the terminating null character. However, the digest value is always represented as a string of hexadecimal numbers that is exactly 64 characters long. It is important to note that even a zero-length object will have a digest, which is the standard SHA-256 NULL hash value.

szContent

A null-terminated string which specifies the MIME content type for the storage object. The content type is determined by the object label and evaluating the contents of the object. It is also possible for the application to explicitly specify the content type of the object when it is created.

dwAttributes

An unsigned integer value that specifies the attributes for the storage object. The object attributes are comprised of one or more bitflags:

Value	Constant	Description
0	WEB_OBJECT_DEFAULT	Default object attributes. This value is used to indicate the object can be modified, or that the attributes for a previously existing object should not be changed.
1	WEB_OBJECT_NORMAL	A normal object that that can be read and modified by the application. This is the default

		attribute for new objects that are created by the application.
2	WEB_OBJECT_READONLY	A read-only object that can only be read by the application. Attempts to modify or replace the contents of the object will fail. Read-only objects can be deleted.
4	WEB_OBJECT_HIDDEN	A hidden object. Objects with this attribute are not returned when enumerated using the WebEnumObjects function. The object can only be accessed directly when specifying its label.

dwObjectSize

An unsigned integer value that specifies the size of the storage object in bytes. The maximum size of an individual object is determined by the storage quota limits established for the account.

stCreated

A SYSTEMTIME structure that specifies the date and time the storage object was created. This value is represented using Coordinated Universal Time (UTC) and is not adjusted for the local time zone.

stModified

A SYSTEMTIME structure that specifies the date and time the storage object was last modified. This value is represented using Coordinated Universal Time (UTC) and is not adjusted for the local time zone. When a storage object is first created, this value will be the same as the object creation time.

Remarks

The object content type will always be in the format **type/subtype** where the type specifies a common media type (e.g.: text, audio, video, etc.) and subtype specifies the specific content. The most common content type for text files is **text/plain**. If the content type is unknown, the default content type is **application/octet-stream**.

Text objects may also optionally include the character encoding as part of the content type. For example, if an object contains UTF-8 encoded text, the content type may be returned as **text/plain; charset=utf-8**. If your application is parsing the content types, you must check if a character encoding was also included in the value. Text objects that do not specify an encoding either contain ASCII or text which uses the system code page. Unicode text will always be stored using UTF-8 encoding.

To adjust the object creation and modification times to account for the local time zone, use the **SystemTimeToTzSpecificLocalTime** function. If you prefer to use FILETIME values, use the **SystemTimeToFileTime** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include cstoools10.h

Unicode: Implemented as Unicode and ANSI versions.

WEB_STORAGE_QUOTA Structure

The **WEB_STORAGE_QUOTA** structure contains information about the current storage account data usage and limits. This structure is used with the **WebGetStorageQuota** function.

```
typedef struct _WEB_STORAGE_QUOTA
{
    DWORD        dwObjects;
    DWORD        dwObjectLimit;
    DWORD        dwObjectSize;
    ULONGLONG    ulBytesUsed;
    ULONGLONG    ulBytesFree;
    ULONGLONG    ulStorageLimit;
} WEB_STORAGE_QUOTA, *LPWEB_STORAGE_QUOTA;
```

Members

dwObjects

An unsigned integer value which specifies the number of storage objects allocated for the account. This value may not exceed the total number of objects specified by the *dwObjectLimit* member.

dwObjectLimit

An unsigned integer value which specifies the maximum number of storage objects that may be created. In addition to the limit on the total amount of storage that may be used, there is a limit on the total number of objects that may be created by all applications.

dwObjectSize

An unsigned integer value which specifies the maximum size of an individual storage object. In addition to a limit on the total amount of storage used and the number of objects created, each object stored by the application cannot exceed this size.

ulBytesUsed

An unsigned 64-bit integer value which specifies the total number of bytes of data allocated for all storage objects. This value may not exceed the total number of bytes of storage available, which is specified by the *ulStorageLimit* member.

ulBytesFree

An unsigned 64-bit integer value which specifies the number of bytes available for the storage of new objects. This value reflects the total amount of available storage across all applications registered with the development account. If this value is zero, your storage account has reached its storage limit.

ulStorageLimit

An unsigned 64-bit integer value which specifies the maximum number of bytes of data storage available. This limit applies to all applications registered with the development account.

Remarks

Storage quota limits are assigned for each SocketTools development account. The **WebGetStorageQuota** function will populate this structure with information about the limits on your account. Accounts that are created with an evaluation license have much lower quota limits than a standard account and should be used for testing purposes only. After the evaluation period has ended, all objects stored using the evaluation license will be deleted.

These values do not represent limits on storage usage by a specific application. Quotas limits

apply to all applications that are registered with the development account, which is identified with the runtime license key passed to the **WebInitialize** function.

If your storage quota has been exceeded, either because the total number of objects or the total bytes of storage has reached their limit, your applications will be unable to create new objects. Your application can continue to access existing objects, regardless of your current quota limits.

To free storage space, use the **WebDeleteObject** function to delete individual storage objects that are no longer needed by your application, or use the **WebResetStorage** function to delete all objects in a container.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header: Include `cstools10.h`

See Also

[WebGetStorageQuota](#)

WEB_STORAGE_TRANSFER Structure

This structure is used by the **WebGetTransferStatus** function to return information about a data transfer in progress.

```
typedef struct _WEB_STORAGE_TRANSFER
{
    TCHAR        szObjectLabel[512];
    DWORD        dwStorageType;
    DWORD        dwBytesTotal;
    DWORD        dwBytesCopied;
    DWORD        dwBytesPerSecond;
    DWORD        dwTimeElapsed;
    DWORD        dwTimeEstimated;
} WEB_STORAGE_TRANSFER, *LPWEB_STORAGE_TRANSFER
```

Members

szObjectLabel

A null-terminated string that specifies the label for the object that is being retrieved, created or replaced.

dwStorageType

An integer value that identifies the storage container type. This will be one of the following values:

Constant	Description
WEB_STORAGE_GLOBAL (1)	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
WEB_STORAGE_DOMAIN (2)	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
WEB_STORAGE_MACHINE (3)	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
WEB_STORAGE_USER (4)	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account

	is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.
--	---

dwBytesTotal

An unsigned integer which specifies the total number of bytes that will be transferred. If the object is being downloaded to the local host, this is the size of the stored object. If the data is being uploaded from the local host to be stored on the server, it is the size of the buffer or local file.

dwBytesCopied

An unsigned integer which specifies the total number of bytes that have been copied.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the data transfer is completed. This is based on the average number of bytes transferred per second.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Whois Protocol Library

Request registration information for an Internet domain name.

Reference

- [Functions](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

File Name	CSWHOV10.DLL
Version	10.0.1468.2518
LibID	A1E831CF-F84F-48B1-9710-D012B50AB6D5
Import Library	CSWHOV10.LIB
Dependencies	None
Standards	RFC 954

Overview

The Whois protocol library provides an interface for requesting registration information for an Internet domain name. When a domain name is registered, the organization that registers the domain must provide certain contact information along with technical information such as the primary name servers for that domain. The library provides an API for requesting that information and returning it to the program so that it can be displayed or processed. This library would be most commonly used to query the Whois server at **whois.internic.net** to obtain information about a specific Internet domain name or an administrative contact at that domain.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). If you are using Visual C++ 6.0 it is required that you have Service Pack 6 (SP6) installed. You should install all updates for your development tools and have the current Windows SDK installed. The minimum recommended version of Visual Studio is Visual Studio 2010.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for your version of the operating system.

This product includes both 32-bit and 64-bit libraries. Native 64-bit CPU support requires the 64-bit version of Windows 7 or later versions of the Windows operating system.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

Whois Protocol Functions

Function	Description
WhoisAsyncConnect	Connect asynchronously to the specified server
WhoisAttachThread	Attach the specified client handle to another thread
WhoisCancel	Cancel the current blocking operation
WhoisConnect	Connect to the specified server
WhoisDisableEvents	Disable asynchronous event notification
WhoisDisableTrace	Disable logging of socket function calls to the trace log
WhoisDisconnect	Disconnect from the current server
WhoisEnableEvents	Enable asynchronous event notification
WhoisEnableTrace	Enable logging of socket function calls to a file
WhoisEventProc	Callback function that processes events generated by the client
WhoisFreezeEvents	Suspend or resume event handling by the calling process
WhoisGetErrorString	Return a description for the specified error code
WhoisGetLastError	Return the last error code
WhoisGetStatus	Return the current client status.
WhoisGetTimeout	Return the number of seconds until an operation times out
WhoisInitialize	Initialize the library and validate the specified license key at runtime
WhoisIsBlocking	Determine if the client is blocked, waiting for information
WhoisIsConnected	Determine if the client is connected to the server
WhoisIsReadable	Determine if there is data available to be read from the server
WhoisRead	Read data returned by the server
WhoisRegisterEvent	Register an event callback function
WhoisSearch	Search for the specified record
WhoisSetLastError	Set the last error code
WhoisSetTimeout	Set the number of seconds until an operation times out
WhoisUninitialize	Terminate use of the library by the application

WhoisAsyncConnect Function

```
HCLIENT WINAPI WhoisAsyncConnect(  
    LPCTSTR LpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwReserved,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **WhoisAsyncConnect** function is used to establish a connection with the server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **WhoisConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Parameters

LpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwReserved

A reserved parameter. This value should always be zero.

hEventWnd

The handle to the event notification window. This window receives messages which notify the client of various asynchronous network events that occur. If this parameter is NULL, then a synchronous (blocking) connection will be established with the server.

uEventMsg

The message identifier that is used when an asynchronous network event occurs. This value should be greater than WM_USER as defined in the Windows header files. If the *hEventWnd* parameter is NULL, this parameter should be specified as WM_NULL.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is INVALID_CLIENT. To get extended error information, call **WhoisGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if

an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
WHOIS_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
WHOIS_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
WHOIS_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
WHOIS_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
WHOIS_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
WHOIS_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To cancel asynchronous notification and return the client to a blocking mode, use the **WhoisDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WhoisConnect](#), [WhoisDisconnect](#), [WhoisInitialize](#), [WhoisSearch](#)

WhoisAttachThread Function

```
DWORD WINAPI WhoisAttachThread(  
    HCLIENT hClient  
    DWORD dwThreadId  
);
```

The **WhoisAttachThread** function attaches the specified client handle to another thread.

Parameters

hClient

Handle to the client session.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is `WHOIS_ERROR`. To get extended error information, call

WhoisGetLastError.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **WhoisAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **WhoisAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **WhoisCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **WhoisUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cswhov10.lib`

See Also

[WhoisCancel](#), [WhoisAsyncConnect](#), [WhoisConnect](#), [WhoisDisconnect](#), [WhoisUninitialize](#)

WhoisCancel Function

```
INT WINAPI WhoisCancel(  
    HCLIENT hClient  
);
```

The **WhoisCancel** function cancels any outstanding blocking operation in the client, causing the blocking function to fail. The application may then retry the operation or terminate the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is WHOIS_ERROR. To get extended error information, call **WhoisGetLastError**.

Remarks

When the **WhoisCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[WhoisIsBlocking](#)

WhoisConnect Function

```
HCLIENT WINAPI WhoisConnect(  
    LPCTSTR lpszRemoteHost,  
    UINT nRemotePort,  
    UINT nTimeout,  
    DWORD dwReserved  
);
```

The **WhoisConnect** function is used to establish a connection with the server.

Parameters

lpszRemoteHost

A pointer to the name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nRemotePort

The port number the server is listening on; a value of zero specifies that the default port number should be used.

nTimeout

The number of seconds that the client will wait for a response from the server before failing the current operation.

dwReserved

A reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is `INVALID_CLIENT`. To get extended error information, call **WhoisGetLastError**.

Remarks

To prevent this function from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **WhoisConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

Requirements

If the function succeeds, the return value is a handle to a client session. If the function fails, the return value is `INVALID_CLIENT`. To get extended error information, call **WhoisGetLastError**.

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cswhov10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WhoisDisconnect](#), [WhoisInitialize](#), [WhoisSearch](#)

WhoisDisableEvents Function

```
INT WINAPI WhoisDisableEvents(  
    HCLIENT hClient  
);
```

The **WhoisDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is WHOIS_ERROR. To get extended error information, call **WhoisGetLastError**.

Remarks

This function affects both event notification and event callbacks. Any outstanding events in the message queue should be ignored by the client application.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[WhoisEnableEvents](#), [WhoisFreezeEvents](#), [WhoisRegisterEvent](#)

WhoisDisableTrace Function

```
BOOL WINAPI WhoisDisableTrace();
```

The **WhoisDisableTrace** function disables the logging of socket function calls to the trace log file.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[WhoisEnableTrace](#)

WhoisDisconnect Function

```
INT WINAPI WhoisDisconnect(  
    HCLIENT hClient  
);
```

The **WhoisDisconnect** function terminates the connection with the server, closing the socket and releasing the memory allocated for the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is WHOIS_ERROR. To get extended error information, call **WhoisGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswhov10.lib

See Also

[WhoisAsyncConnect](#), [WhoisConnect](#), [WhoisUninitialize](#)

WhoisEnableEvents Function

```
INT WINAPI WhoisEnableEvents(  
    HCLIENT hClient,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **WhoisEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **WhoisRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hClient

Handle to the client session.

hEventWnd

Handle to the event notification window. This window receives a user-defined message which specifies the event that has occurred. If this value is NULL, event notification is disabled.

uEventMsg

An unsigned integer which specifies the user-defined message that is sent when an event occurs. This parameter's value must be greater than the value of WM_USER.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is WHOIS_ERROR. To get extended error information, call **WhoisGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the client handle. One or more of the following event identifiers may be sent:

Constant	Description
WHOIS_EVENT_CONNECT	The connection to the server has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
WHOIS_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
WHOIS_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
WHOIS_EVENT_WRITE	The client can now write data. This notification is sent after a

	connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
WHOIS_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
WHOIS_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To cancel asynchronous notification and return the client to a blocking mode, use the **WhoisDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswhov10.lib

See Also

[WhoisDisableEvents](#), [WhoisFreezeEvents](#), [WhoisRegisterEvent](#)

WhoisEnableTrace Function

```
BOOL WINAPI WhoisEnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **WhoisEnableTrace** function enables the logging of socket function calls to a file.

Parameters

lpszTraceFile

A pointer to a string which specifies the name of the trace log file. If this parameter is NULL or an empty string, the default file name **cstrace.log** is used. The file is stored in the directory specified by the TEMP environment variable, if it is defined; otherwise, the current working directory is used.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Constant	Description
TRACE_INFO	All function calls are written to the trace file. This is the default value.
TRACE_ERROR	Only those function calls which fail are recorded in the trace file.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are recorded in the trace file.
TRACE_HEXDUMP	All functions calls are written to the trace file, plus all the data that is sent or received is displayed, in both ASCII and hexadecimal format.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the tracing library **cstrcv10.dll** cannot be loaded on the local system.

Remarks

When trace logging is enabled, the file is opened, appended to and closed for each socket function call. This makes it possible for an application to append its own logging information, however care should be taken to ensure that the file is closed before the next network operation is performed. To limit the size of the log files, enable and disable logging only around those sections of code that you wish to trace.

Note that the TRACE_HEXDUMP trace level generates very large log files since it includes all of the data exchanged between your application and the server.

Trace function logging is managed on a per-thread basis, not for each client handle. This means that all SocketTools libraries and components share the same settings in the current thread. If you

are using multiple SocketTools libraries or components in your application, you only need to enable logging once.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WhoisDisableTrace](#)

WhoisEventProc Function

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

The **WhoisEventProc** function is an application-defined callback function that processes events generated by the calling process.

Parameters

hClient

The handle to the client session.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the **WhoisRegisterEvent** function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **WhoisRegisterEvent** function. The **WhoisEventProc** function is a placeholder for the application-defined function name.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswhov10.lib

See Also

[WhoisDisableEvents](#), [WhoisEnableEvents](#), [WhoisFreezeEvents](#), [WhoisRegisterEvent](#)

WhoisFreezeEvents Function

```
INT WINAPI WhoisFreezeEvents(  
    HCLIENT hClient,  
    BOOL bFreeze  
);
```

The **WhoisFreezeEvents** function is used to suspend and resume event handling by the client.

Parameters

hClient

Handle to the client session.

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is WHOIS_ERROR. To get extended error information, call **WhoisGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all client events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the client has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[WhoisDisableEvents](#), [WhoisEnableEvents](#), [WhoisRegisterEvent](#)

WhoisGetErrorString Function

```
INT WINAPI WhoisGetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT cbDescription  
);
```

The **WhoisGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the **WhoisGetLastError** function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The last-error code for which a description is returned.

lpszDescription

A pointer to the buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

cbDescription

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is 0, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: `cstools10.h`

Import Library: `cswhov10.lib`

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WhoisGetLastError](#), [WhoisSetLastError](#)

WhoisGetLastError Function

```
DWORD WINAPI WhoisGetLastError();
```

Parameters

None.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **WhoisSetLastError** function. The Return Value section of each reference page notes the conditions under which the function sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **WhoisGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **WhoisSetLastError(0)** when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or WHOIS_ERROR. Those functions which call **WhoisSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[WhoisGetErrorString](#), [WhoisSetLastError](#)

WhoisGetStatus Function

```
INT WINAPI WhoisGetStatus(  
    HCLIENT hClient  
);
```

The **WhoisGetStatus** function returns the current status of the client session.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is WHOIS_ERROR. To get extended error information, call **WhoisGetLastError**.

Remarks

The **WhoisGetStatus** function returns a numeric code which identifies the current state of the client session. The following values may be returned:

Value	Constant	Description
1	WHOIS_STATUS_IDLE	The client is current idle and not sending or receiving data.
2	WHOIS_STATUS_CONNECT	The client is establishing a connection with the server.
3	WHOIS_STATUS_READ	The client is reading data from the server.
4	WHOIS_STATUS_WRITE	The client is writing data to the server.
5	WHOIS_STATUS_DISCONNECT	The client is disconnecting from the server.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified client session.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[WhoisIsBlocking](#), [WhoisIsConnected](#), [WhoisIsReadable](#)

WhoisGetTimeout Function

```
INT WINAPI WhoisGetTimeout(  
    HCLIENT hClient  
);
```

The **WhoisGetTimeout** function returns the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

Return Value

If the function succeeds, the return value is the timeout period in seconds. If the function fails, the return value is WHOIS_ERROR. To get extended error information, call **WhoisGetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswhov10.lib

See Also

[WhoisSetTimeout](#)

WhoisInitialize Function

```
BOOL WINAPI WhoisInitialize(  
    LPCTSTR lpszLicenseKey,  
    LPINITDATA lpData  
);
```

The **WhoisInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpData

Pointer to an INITDATA data structure. This parameter may be NULL if the initialization data for the library is not required.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **WhoisGetLastError**. All other client functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must insure that this function is called with a valid license key.

If the *lpData* argument is specified, it must point to an INITDATA structure which has been initialized by setting the *dwSize* member to the size of the structure. All other structure members should be set to zero. If the function is successful, then the INITDATA structure will be filled with identifying information about the library.

Although it is only required that **WhoisInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **WhoisUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **WhoisInitialize** or **WhoisUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WhoisAsyncConnect](#), [WhoisConnect](#), [WhoisDisconnect](#), [WhoisUninitialize](#)

Copyright © 2023 Catalyst Development Corporation. All rights reserved.

WhoisIsBlocking Function

```
BOOL WINAPI WhoisIsBlocking(  
    HCLIENT hClient  
);
```

The **WhoisIsBlocking** function is used to determine if the client is currently performing a blocking operation.

Parameters

hClient

Handle to the client session.

Return Value

If the client is performing a blocking operation, the function returns TRUE. If the client is not performing a blocking operation, or the client handle is invalid, the function returns FALSE.

Remarks

Because a blocking operation can allow the application to be re-entered (for example, by pressing a button while the operation is being performed), it is possible that another blocking function may be called while it is in progress. Since only one thread of execution may perform a blocking operation at any one time, an error would occur. The **WhoisIsBlocking** function can be used to determine if the client is already blocked, and if so, take some other action (such as warning the user that they must wait for the operation to complete).

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[WhoisCancel](#), [WhoisGetStatus](#), [WhoisIsConnected](#), [WhoisIsReadable](#)

WhoisIsConnected Function

```
BOOL WINAPI WhoisIsConnected(  
    HCLIENT hClient  
);
```

The **WhoisIsConnected** function is used to determine if the client is currently connected to a server.

Parameters

hClient

Handle to the client session.

Return Value

If the client is connected to a server, the function returns a non-zero value. If the client is not connected, or the client handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswhov10.lib

See Also

[WhoisGetStatus](#), [WhoisIsBlocking](#), [WhoisIsReadable](#)

WhoisIsReadable Function

```
BOOL WINAPI WhoisIsReadable(  
    HCLIENT hClient,  
    INT nTimeout,  
    LPDWORD lpdwAvail  
);
```

The **WhoisIsReadable** function is used to determine if data is available to be read from the server.

Parameters

hClient

Handle to the client session.

nTimeout

Timeout for server response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the client can read data from the server within the timeout period, the function returns a non-zero value. If the client cannot read any data, the function returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[WhoisGetStatus](#), [WhoisIsBlocking](#), [WhoisIsConnected](#)

WhoisRead Function

```
INT WINAPI WhoisRead(  
    HCLIENT hClient,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **WhoisRead** function reads the specified number of bytes from the client socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hClient

Handle to the client session.

lpBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the server has closed the connection and there is no more data available to be read. If the function fails, the return value is WHOIS_ERROR. To get extended error information, call **WhoisGetLastError**.

Remarks

When **WhoisRead** is called and the client is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[WhoisIsBlocking](#), [WhoisIsReadable](#), [WhoisSearch](#)

WhoisRegisterEvent Function

```
INT WINAPI WhoisRegisterEvent(  
    HCLIENT hClient,  
    UINT nEventId,  
    INETEVENTPROC LpfnEvent,  
    DWORD_PTR dwParam  
);
```

The **WhoisRegisterEvent** function registers a callback function for the specified event.

Parameters

hClient

Handle to client session.

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One or more of the following values may be used:

Constant	Description
WHOIS_EVENT_CONNECT	The connection to the server has completed.
WHOIS_EVENT_DISCONNECT	The server has closed the connection to the client. The client should read any remaining data and disconnect.
WHOIS_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
WHOIS_EVENT_WRITE	The client can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the client is in asynchronous mode.
WHOIS_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The client application may attempt to retry the operation, or may disconnect from the server and report an error to the user.
WHOIS_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

lpfnEvent

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **WhoisEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Remarks

The **WhoisRegisterEvent** function associates a callback function with a specific event. The event handler is an **WhoisEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is WHOIS_ERROR. To get extended error information, call **WhoisGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[WhoisDisableEvents](#), [WhoisEnableEvents](#), [WhoisEventProc](#), [WhoisFreezeEvents](#)

WhoisSearch Function

```
INT WINAPI WhoisSearch(  
    HCLIENT hClient,  
    LPCTSTR lpszKeyword,  
    UINT nSearchType  
);
```

The **WhoisSearch** function submits the specified keyword to the server.

Parameters

hClient

Handle to the client session.

lpszKeyword

Points to a string which specifies the query keyword. It may be a handle, name or mailbox.

nSearchType

The type of search being performed. One of the following values may be used:

Constant	Description
WHOIS_SEARCH_ANY	Search for any record that matches the specified keyword.
WHOIS_SEARCH_HANDLE	Search only for handles that match the specified keyword.
WHOIS_SEARCH_MAILBOX	Search only for mailboxes that match the specified keyword.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is WHOIS_ERROR. To get extended error information, call **WhoisGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

Unicode: Implemented as Unicode and ANSI versions.

See Also

[WhoisAsyncConnect](#), [WhoisConnect](#), [WhoisIsReadable](#), [WhoisRead](#)

WhoisSetLastError Function

```
VOID WINAPI WhoisSetLastError(  
    DWORD dwErrorCode  
);
```

The **WhoisSetLastError** function sets the error code for the current thread. This function is typically used to clear the last error by passing a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_CLIENT or WHOIS_ERROR. Those functions which call **WhoisSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the **WhoisGetLastError** function. The use of **WhoisGetLastError** is optional; an application can call the function to determine the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[WhoisGetErrorString](#), [WhoisGetLastError](#)

WhoisSetTimeout Function

```
INT WINAPI WhoisSetTimeout(  
    HCLIENT hClient,  
    UINT nTimeout  
);
```

The **WhoisSetTimeout** function sets the number of seconds the client will wait for a response from the server. Once the specified number of seconds has elapsed, the function will fail and return to the caller.

Parameters

hClient

Handle to the client session.

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is WHOIS_ERROR. To get extended error information, call **WhoisGetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Import Library: cswhov10.lib

See Also

[WhoisGetTimeout](#)

WhoisUninitialize Function

```
VOID WINAPI WhoisUninitialize();
```

The **WhoisUninitialize** function terminates the use of the library.

Parameters

There are no parameters.

Return Value

None.

Remarks

An application is required to perform a successful **WhoisInitialize** call before it can call any of the other the library functions. When it has completed the use of library, the application must call **WhoisUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There must be a call to **WhoisUninitialize** for every successful call to **WhoisInitialize** made by a process. In a multithreaded environment, operations for all threads in the client are terminated.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: cstools10.h

Import Library: cswhov10.lib

See Also

[WhoisDisconnect](#), [WhoisInitialize](#)

Whois Protocol Data Structures

- INITDATA

INITDATA Structure

This structure contains information about the SocketTools library when the initialization function returns.

```
typedef struct _INITDATA
{
    DWORD        dwSize;
    DWORD        dwVersionMajor;
    DWORD        dwVersionMinor;
    DWORD        dwVersionBuild;
    DWORD        dwOptions;
    DWORD_PTR    dwReserved1;
    DWORD_PTR    dwReserved2;
    TCHAR        szDescription[128];
} INITDATA, *LPINITDATA;
```

Members

dwSize

Size of this structure. This structure member must be set to the size of the structure prior to calling the initialization function. Failure to do so will cause the function to fail.

dwVersionMajor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the major version number for the library.

dwVersionMinor

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the minor version number for the library.

dwVersionBuild

This member must be initialized to a value of zero prior to calling the initialization function. When the function returns, it will contain the build number for the library.

dwOptions

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved1

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

dwReserved2

This member must be initialized to a value of zero prior to calling the initialization function. This member is reserved for future use.

szDescription

A null terminated string which describes the library.

Requirements

Minimum Desktop Platform: Windows 7 (Service Pack 1)

Minimum Server Platform: Windows Server 2008 R2 (Service Pack 1)

Header File: ctools10.h

Unicode: Implemented as Unicode and ANSI versions.
